USENIX

HIGH-SPEED NETWORKING SYMPOSIUM

SUMMER 1994

U
US
USE
U NIX
USENIX

# High-Speed Networking

**Symposium Proceedings**

**Oakland, California**
**August 1-3, 1994**

For additional copies of these proceedings write:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is $15 for members and $20 for nonmembers.
Outside the U.S.A. and Canada, please add
$9 per copy for postage (via air printed matter).

ISBN 1-880446-63-4

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

# Proceedings of the 1994 USENIX Symposium on High-Speed Networking

August 1-3, 1994
Oakland, California, USA

# High-Speed Networking Symposium

**August 1-3, 1994**
**Oakland, California**

**Table of Contents**

# Modular Communication Subsystem Implementation using a Synchronous Approach

Claude Castelluccia Walid Dabbous

*INRIA, 2004 Route des Lucioles*
*BP-93, 06902 Sophia Antipolis Cedex, FRANCE*
*e-mail: Firstname.Lastname@inria.fr*

**Abstract**

The lack of flexibility and performance of current communication subsystems has led researchers to look for new protocol architectures. A new design philosophy, flexible and efficient, referred to in the literature as *"function-based communication model"* is emerging and seems to be very promising. It consists of designing application-tailored communication subsystems adapted to the specific requirements of a given application. The flexibility of such a solution leads to very efficient implementations integrating only required functionalities.

In this paper, we propose a flexible model which uses a synchronous language to synthesize communication subsystems from functional building blocks. We prove the feasibility of our approach by implementing a data transfer protocol using Esterel, a synchronous language. Communication subsystem specifications in our model are very modular; they are composed of parallel modules, implementing the different functionalities of the communication subsystem, which synchronize and communicate using signals. The Esterel compiler generates from this parallel specification a sequential automaton by resolving resource conflicts. The design flexibility of our approach is demonstrated; modules are selected according the application requirements and compiled to generate an integrated implementation.

## 1 Introduction

### 1.1 Motivation

The lack of flexibility and performance of current communication subsystems has led researchers to look for new protocol architectures [CT90, HP88, SSS+93, OP91, ANMD93].

On one hand, the avalanche of new applications with new requirements and the rapid change in network technologies place new demands on communication services. Integrating all possible services into a monolithic general-purpose communication subsystem seems to be an unrealistic solution, because it would be practically difficult to implement and maintain.

On the other hand, the inefficiency of current layered architectures is admitted [SSS+93, BSS92, AP92b, HP88]. There are several reasons for this lack of performance, among them the replication of functions in different layers, the overhead of control messages and their inadequacy for the parallelization of protocol processing.

A new design philosophy, flexible and efficient, referred to in the literature as "*function-based communication model*" is emerging and seems to be very promising. It consists of dynamically developing application-tailored subsystems which adapt to the specific requirements of a given application. The flexibility of such a solution leads to very efficient implementations integrating only required functionalities.

The generation of "tailored" subsystems addresses others important issues and questions that have to be studied concurrently : (1) How to specify the application requirements, (2) How to select the right mechanisms to suit application needs, (3) How to generate and implement the "tailored" communication subsystem.

In this paper, we mainly focus on the third issue.

### 1.2 Goals

The idea of "function-based" communication subsystems has been extensively discussed in the research literature but, to our knowledge, no working and efficient experiments have been performed.

In this paper, we present a practical solution for combining elementary building blocks in order to generate and implement a "function-based" communication subsystem using synchronous languages. Our approach is general: it can be applied to all the communication functionalities required for a selected application.

Our final goal is to propose a modular and yet efficient way to implement communication subsystems. However in this paper, we mainly focus on the modularity aspects. We propose a highly structured and flexible communication subsystem architecture that generates integrated implementations. Our approach is based on the use of synchronous languages to synthesize communication subsystems from basic building blocks. In order to validate our approach, we implemented a data transfer protocol by the synthesis of elementary building blocks. We thus show that synchronous languages such as Esterel may be used to express the protocol control part, and to generate an optimal (in terms of task scheduling ) automaton for the protocol execution.

The rest of the paper is organized as follows. Section 2 presents the benefits of the function-based approach. Section 3 briefly introduces synchronous languages and shows their adequacy for flexible subsystem synthesis. In section 4, we prove the feasibility of our approach by implementing a data transfer protocol, and analyze the results obtained. Section 5 offers conclusions based on our experience and suggests direction for future research.

## 2 Communication subsystems design

New applications (e.g. audio and video conferencing, shared whiteboard, supercomputer visualization etc.) with specific communication requirements are being considered. Depending on the application, these requirements may be one or more of the following: (1) high bit rates, (2) low jitter data transfer, (3) the simultaneous exchange of multiple data streams with different "type of service" such as audio, video and textual data, (4) a reliable multicast data transmission service, (5) low latency transfer for RPC based applications, (6) specific presentation encodings, etc.

The above requirements imply the necessity of revising the service model in order to fulfill the specific application needs. The applications must be able to specify the control mechanisms of a complete communication subsystem and not only the parameters of a single transport service. Recent research activities in this domain propose the synthesis of the so-called "communication subsystems", tailored to provide the service required by the application, from "building blocks" implementing elementary protocol functions such as: flow control, error control, and connection management (e.g. [SSS+93, AP92b, BSS92, OP91]). The synthesis of "fine grain" protocol functions should replace the coarse grain protocol choice (e.g. TCP or UDP).

However, these synthesis activities are focused on transport level functions, and do not consider the application synchronization and data presentation requirements. We argue that the integration of all the application communication requirements (including transmission control, synchronization and presentation encoding), in a single optimized protocol graph will result in increased performance. This is in line with the ALF (Application Level Framing) architecture proposed by Clark and Tennenhouse in [CT90].

This approach puts the application inside the "control loop". We still need to define how the application level parameters will be mapped onto network parameters and control functions in order to build specialized communication subsystems. The most promising approach in our opinion consists of the following combination: the use of a suitable language to express the application synchronization requirements (or the application dynamics) and the design and implementation of a tool (a compiler) that derives the complete communication subsystem automatically based on a set of optimized building blocks implementing the protocol functionalities. These functions should be combined in a way that minimizes the memory access cost.

In this paper, we mainly focus on the synthesis of the communication subsystem from functional building blocks; we are not detailing the application-communication integration part. In order to test the feasibility of the communication subsystem synthesis we used the Esterel language to describe the control part and to generate an implementation of a TCP-like protocol based on functional building blocks. The generality of our approach extends to the support of the complete subsystem's implementation and not only to transport functionalities.

## 3 Synchronous languages

In this section, we review the basic concepts of synchronous languages, and in particular those of Esterel, and show why they are appropriate to our final goal [Ber89].

Programs can basically be divided into three classes : (1)transformational programs that compute results from a given set of inputs, (2) interactive programs which interact at their own speed with users or other programs and (3) reactive programs that interact with the environment, at a speed determined by the environment, not by the program itself.

Synchronous languages were specifically designed for implementing reactive systems. The Esterel language is one example [BdS91, BG89]; others include languages such as Lustre, Signal, Sml and Statecharts.

Protocols are good examples of reactive systems; they can be seen as "black boxes", activated by input events (such as incoming packets) and reacting by producing output events (such as outgoing packets).

Esterel programs are composed of parallel modules, which communicate and synchronize using signals. The output signal of a module is broadcast within the whole program and can be tested for presence and valued by any other modules. This communication mechanism provides a lot of design flexibility, because modules can be added, removed or exchanged without perturbing the overall system. A module is defined by its inputs (the signals that activate it, they can potentially be modified by the receiving module), sensors (input signals used only for consultation, they can not be modified) and outputs (signals emitted). The inputs of a module can be the outputs of another one (modules executed sequentially) or external inputs (such as incoming packets). The design of an Esterel program is then performed by combining and synchronizing the different elementary modules using their input, sensor and output signals. Synchronous languages are used to implement the control part of a program, the computational and data manipulation parts are performed by functions, implemented in another language (C for example).
Data declaration are encapsulated, so that only the visible interface declarations must be provided in Esterel: type/constant/function/procedure names. These declarations can then be freely performed independently of the Esterel program design. It will be linked with the automaton generated in

the last phase, when executable code is produced. This separation between the control and data manipulation makes the integration of new implementation techniques such as *Integrated Layer Processing*[CT90, CJRS89] easier and allow to defer data handling to full-scale existing compilers. However the use of opaque declaration precludes reasoning on actual values prior to the linking phase, since the meaning of data operation is left uninterpreted.

Esterel makes the assumption of perfect synchrony : program reactions can not overlap. There is no possibility of activating a system while it is still reacting to the current activation. This assumptions makes Esterel programs deterministic, since behavior are then reproducible; the generated automaton can then be tested for correctness using validation tools [RdS90].

At compile time, the Esterel program is translated into a sequential finite automaton; the code of the different modules is sequentialized according the program concurrency and synchronization specifications.

However the synchronous approach cannot be considered as a stand-alone solution, principally because the synchronous assumption is not a valid one in the real implementation world. A so-called "Execution Machine" is required [AMP91]. This machine is aimed at interfacing the asynchronous environment to the synchronous automaton. It collects the inputs and outputs and activates the automaton only when it is not executing; the "synchronous assumption" is then respected.

## 4 Communication Subsystem implementation with Esterel: a study case

In this section, we describe the implementation of a data transfer protocol using Esterel. The specification of the protocol that we implemented is very similar to that of TCP protocol (we omit however the connection establishment and termination phases). We address how to design the building blocks and how to combine them to generate the required protocol. The goals of this case study were to test the validity of our approach and to give an insight on building-blocks contents.

Reusability and flexibility are our main goals here. The building blocks must be designed so that they are meaningful to the designers and so that changes in the protocol specification only induce local changes in the architecture and the code.

Communication subsystems are mainly structured in 3 parts [KTZ92]:

- the send module, that handles outgoing frames
- the receive module, that processes incoming frames
- the connection module, that handles connection variables and states

Each of these components can be decomposed into finer grain modules. The protocol functions are considered as atomic and their dependencies are defined by their input, sensor and output signals.

### 4.1 Building Blocks Description

We implemented this example incrementally in order to satisfy the modularity property that we are aiming for. We started from a simple protocol and added modules step by step until we accomplished all required functionalities. Following the precepts of Object Oriented Programming we followed the rule *one functionality-one module*. An overview of the whole subsystem is shown in figure 1 . For clarity purposes, only the principal modules have been described and displayed. Our data-transfer protocol is structured in 3 main concurrent modules :

## The Send Module

This module is composed of 2 concurrent submodules:

The *Input_Handler* module receives data from the application. If enough space is left in the internal buffer, they are copied to it and a "Try-to-Snd" signal is broadcast, otherwise incoming data are unauthorized until an acknowledgment signal frees up some buffer space.

The *Emission_Handler* module transmits packets on the network. It can received two types of inputs: the "Try-to-Send" input will try to send packets by evaluating the congestion window size, the silly window avoidance algorithm and the number of bytes waiting to be sent; it may then send one or several packets. The "Send-Now" input will force the sending of a packet even though the regular sending criteria are not satisfied (this is used to acknowledge data for example). If the module decides to send packets, the checksum is performed and the header is completed.

## The Receive Module

This module processes the incoming packets. It is composed of several submodules that can be executed concurrently:

When a packet is received, its header is scanned by the *Scan-Handler* module and all the header fields are broadcast.

The *Validate-Packet* module waits for "Checksum" and "Off-bit" signals to validate the incoming data (checksum and Off-bit field conformance tests are performed). If the packet is non-valid it is rejected at this point, otherwise the data are processed (the acknowledgment field is processed concurrently by the *Connection module*).

A test is performed (in the *Process-Path-Check* module) to decide whether the packet should follow the "header prediction" path (*header-predictor* module) or normal path (*Normal-Process-Handler module*).
In the normal path, the data are processed and delivered to the application; the flow control parameters are updated.
In the fast path, two cases are possible:
(1)A pure acknowledgment packet is received. Buffer spaces are fred up and the sequence number of the next data to be acknowledged is updated. (2)A pure in-sequence data packet is received. The data is directly delivered to the application. The sequence number of the next expected data is updated.

## The Connection module

This module is composed of the following submodules that are executed concurrently: the *RTT_Manager* module, the *Timer_Handler* module, the *Window_Manager* module and the *Acknowledgment_Manager* module.

The *RTT_manager* module computes the round trip time of the connection. When a packet is emitted, no other packet belonging to this connection is in transit and we are not in a retransmission phase then a timer is started. When this packet is acknowledged, then the timer is stopped and a new RTT value is computed.

The *Window_manager* updates (concurrently) the congestion window and the send window (corresponding to an evaluation of the space left in the receiving buffer of the remote host). When an acknowledgment signal is received (from the *Acknowledgment_Manager* module), the *Window_manager* increases the congestion window either linearly or exponentially according the slow-start algorithm, and the sending window is either update to the value of the "Win" field (emitted by the*Scan_Handler* module) or decreased by the number of bytes acknowledged. If a signal corresponding to a window shrink request (from the *Timer_Handler* module) is received, the congestion window is set to 512 bytes (value of the Maximum Segment Size).

The *Acknowledgment_Manager* module handles the acknowledgment information received from the incoming packet. If the acknowledgment sequence number (which corresponds to the last bytes received by the remote host) received is greater than the latest bytes sent or less than the last already acknowledged bytes then it is ignored, otherwise the acknowledged value is updated.

The *Timer_Handler* module manages the different timers. When a packet is emitted and no other packets are in transit, the Retransmission timer is set. When this packet is acknowledged and other packets are in transit it is restarted, otherwise it is reset. If the timer expires before the acknowledgment of this packet is received, retransmission and window-shrink requests are emitted.

All the modules just described have been implemented in Esterel and compiled into an automaton.

## 4.2 Application Programming

The protocol generated was then used to implement a file transfer application in order to validate the conformance of our protocol implementation with its specification. In our model, the protocol is implemented at the user level and linked with the application. The protocol is completely integrated with the application; actually the protocol is a task of the application. The application and the protocol can then share the same memory space. We implemented this interaction using Light-Weight Processes which allow the definition of multiple tasks within a single UNIX process and provide light-weight context switching between tasks.

Our application (contained in one process) is composed of 4 tasks :

- the application task : This is the task implementing the application. It notifies the protocol task when some data have to be sent.
- the I/O task : This is the task responsible for the incoming packets. When a new packet from the network is received, the I/O task notifies the protocol task that an incoming packet is waiting to be processed.
- the protocol task : This task processes the outgoing data (coming from the application) and the incoming frame (coming from the I/O task) according to the synthesized protocol. It performs what we called earlier the Esterel "Execution Machine".
- the timer task : This task manages the timers used by the protocols and the application.

All these tasks have the same execution priority, except the I/O task which has an higher one to avoid loss of incoming packet. A task with higher priority may preempt the others. Tasks with the same priority are executed on a round-robin discipline: a task is executed until is done or blocked on an I/O, the following task on the list is then executed.

All these tasks synchronize with each other. When a task has nothing to do, it sleeps. When another task needs its assistance, it is woken up. For example, when the protocol task has neither outgoing data nor incoming packet to process, it sleeps. When a packet arrives, the I/O task awakes it. The synchronization is then very natural.

## 4.3 Analysis of the Results

### 4.3.1 Esterel Compilation Phase

As explained in section 3, Esterel programs are composed of parallel modules, which communicate and synchronize using signals. An Esterel module gets activated on the reception of one or several signals (input signals), executes some actions and emits one or several signals (output signals). It uses local variables (invisible to the others modules) and communicates with other modules using signals (global variables).

The Esterel compiler generates from this parallel specification a sequential automaton by resolving resource conflicts. It assigns a code to every elementary action (e.g. assignment, test) of each module in the program text, and serializes these codes such that actions are always performed on the latest emitted signals within an instant. For example, if a *module A* needs the value of a signal *sig1* for its internal processing and *sig1* is modified and emitted in the same instant by *module B*, then Esterel compiler serializes theses two components such that the code modifying and emitting *sig1* be scheduled before the code using its values. If no schedule can be found, the program is rejected. This is what is called a causality error. Signals do not appear in the automaton. They are implemented as global variables, available to all modules that declare them as inputs or outputs. Emitting a signal consists of updating the corresponding global variable, reading a signal consists of accessing its value.

### 4.3.2 Modularity issues

The modularity of our approach has been demonstrated and illustrated by our case study. In fact we implemented our protocol incrementally and ran it at each step to test its correctness. As a result of this programming style, the program obtained is very modular; we can easily remove or exchange modules. For example by simply discarding the *Window_Handler* submodule we synthesize a sliding window data transfer protocol. If we set the window size to one packet, we implement a Stop-and-Go protocol. The window flow control can also be exchanged by a rate control mechanism by simply modifying the *Emission_Handler* module. The isolation of the different functionalities into separated and concurrent modules leads to very modular structures.

### 4.3.3 Performance issues

Performance was not the main goal of this paper. In fact, in this work we mainly focused on flexibility and modularity aspects.

However in this section, we show that the use of the synchronous language, Esterel, does not necessarily imply bad performances. To demonstrate this point, we compared the C code generated by the Esterel compiler with a handcoded BSD-TCP version of the University of California.

Since we implemented our case-study at the user-level on UDP, we believe that throughput or latency comparisons are not good performance indicators; the overhead due to the OS support is too difficult to evaluate. We therefore compare the structure of the C code of both implementations.

Following the analysis approach described in [CJRS89], we focused on the input processing analysis, which seems to be the bottleneck in protocol processing.
Our protocol is not a fully functional TCP, so it is dangerous to compare the input processing costs too closely and use comparison benchmarks such as instruction count numbers. We instead compare the sequence of actions executed on packet input paths of both implementations. We believe that this coarse grain comparison should give a fair indication of the performance of the code generated by Esterel.

In both implementation, when a packet is received its checksum and offset flag are verified, the protocol control block is retrieved, some variables are initialized and the test for the header prediction is performed. If this test is positive the header prediction algorithm is performed, otherwise the normal processing path is followed. In the normal processing path, the data are possibly trimmed and flow control variables updated.

We observed that the actions, their order of execution and the code executed on packet reception are very similar for both implementations. No specific overheads appear in the Esterel C code. Within a state, the code produced by Esterel is very compact.
However despite this similarity, some differences exist in the way these two programs are structured; The BSD-TCP implementation is "Action Oriented", while the Esterel implementation is "State Oriented".
In fact, the BSD code is composed of a sequence of actions preceded by their condition of activation. This sequence of <condition, action > is processed each time an input event occurs. For example, the condition corresponding to the header prediction algorithm is tested each time the input processing procedure is invoked. This is not always a desirable and time optimal feature.
The Esterel code is structured into states. All these states are distinct and only contain the actions possible in that particular state. For example, in the retransmission phase state the code corresponding to the header prediction algorithm does not appear, because no header prediction is possible in that state.

The receive paths in the Esterel program should then be more time efficient, because no unnecessary tests are performed. However the price to pay for this performance improvement is a code size increase.
The code size of the BSD-TCP is optimum, code sharing is performed whenever possible. In the Esterel program, no code duplication is performed . So if an action is executed in several states, its code is duplicated in each of those states, which can lead to very large compiled programs. As a matter of fact, our Esterel program which is about 10000 lines long produces an 11 states-automaton; the compiled code size of this automaton is about 100 kbytes, 4 times larger than the BSD one.

These preliminary results are very encouraging.
Our analysis shows that there is no inherent reason to believe that the Esterel code should not perform at least as well as the current one. In addition there is still the possibility of tuning the Esterel code (or modifying the Esterel compiler) to combine these approaches in a more optimal way.
We are now expecting to gain performance from the exploitation of the application-level knowledge.

## 5 Conclusions & Future work

The next generation communication subsystems need to be flexible to adapt to the rapid change of application needs and network technologies. In this paper, we proposed a framework using a synchronous language, Esterel, to synthesize communication from functional building-blocks. Esterel language programming style, which consists of breaking a program into components communicating via signals, provides the flexibility and efficiency we are aiming for. It also facilitates the integration of new design principles and implementation techniques, such as ALF/ILP, because of the separation made between the control and the data manipulation paths of a program. The viability and modularity of our framework has been established by our case study.

Our final goal is to propose a modular and yet efficient implementation of a communication subsystem framework. The modularity of our approach has been fully established by the work described in this paper. Efficiency aspects have now to be considered in more details. To achieve this next goal, we are currency addressing some important issues, among them :

- How to integrate efficiently new architectural and implementation concepts such as *Application Level Framing* (ALF) and *Integrated Layer Processing* (ILP) [CT90, GPSV91, AP92a]
- How to exploit application-level knowledge to design optimal communication subsystems

- What OS support is required for efficient and flexible subsystem design [MRA87, MJ93, HP88, MB92, MJ93, ANMD93]

These issues are not specific to our approach but common to high performance subsystem design framework. However we believe that if the right OS support is provided, modular frameworks, which synthesize tailored and optimal subsystems, should perform better than monolithic general purpose architectures.

## Acknowledgment

We would like to thank Isabelle Christment and Ellen Siegel for valuable suggestions and comments.

## Biographies

**Claude Castelluccia** received the B.S. degree from the University of Technology of Compiegne, France, in computer sciences in 1989 and the M.S. degree from Florida Atlantic University in electrical engineering in 1991. He is currently working on a Phd at INRIA (Institut National de Recherche en Informatique et Automatisme), France, on high performance communication protocol design.

## References

[AMP91] C. André, J.P. Marmorat, and J.P. Paris. Execution machines for esterel. In *European Control Conference, Grenoble*, July 1991.

[ANMD93] Chandramohan A.Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D.Lazowska. Implementing network protocols at user level. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols*, September 1993.

[AP92a] Mark B. Abbott and Larry L. Peterson. Automated integration of communication protocol layers. Technical Report TR 92-24, Department of Computer Science, University of Arizona, December 1992.

[AP92b] Mark B. Abbott and Larry L. Peterson. A language-based approach to protocol implementation. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols*, August 1992.

[BdS91] Frédéric Boussinot and Robert de Simone. The ESTEREL language. Technical Report 1487, INRIA U.R. Sophia-Antipolis, July 1991.

[Ber89] Gérard Berry. Real-time programming: Special purpose or general purpose languages. In *Information Processing IFIP Conference, Elsevier Science Publishers, B.V. North Holland*, September 1989.

[BG89] Gérard Berry and Georges Gonthier. Incremental development of an hdlc protocol in esterel. Technical Report 1031, INRIA U.R. Sophia-Antipolis, May 1989.

[BSS92] Donald F. Box, Douglas C. Schmidt, and Tatsuya Suda. ADAPTIVE - an object-oriented framework for flexible and adaptive communication protocols. In *Proceedings of the Fourth IFIP Conference on High Performance Networking*, December 1992.

[CJRS89] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of tcp processing overhead. In *IEEE Communications Magazine*, June 1989.

[CT90] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols*, pages 200–208, September 1990.

[GPSV91] Per Gunningberg, Craig Partridge, Teet Sirotkin, and Bjorn Victor. Delayed evaluation of gigabit protocols. In *Proceedings of the Second MultiG Workshop*, June 1991.

[HP88] N. Hutchinson and L. Peterson. Design of the x-kernel. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols*, pages 65–75, August 1988.

[KTZ92] O.G. Koufopavlou, A.N. Tantawy, and M. Zitterbart. Analysis of TCP/IP for high performance parallel implementations. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, pages 576–585, February 1992.

[MB92] Chris Maeda and Brian N. Bershad. Networking performance for mocrokernels. In *Proceedings of the third Workshop on Workstation Operating Systems*, April 1992.

[MJ93] Steven McCanne and Van Jacobson. A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX Conference*, January 1993.

[MRA87] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, November 1987.

[OP91] S. W. O'Malley and L. L. Peterson. A highly layered architecture for high-speed networks. In *Protocols for High-Speed Networks II IFIP*, 1991.

[RdS90] V. Roy and R. de Simone. Auto and autograph. In *Proceedings of Workshop on Computer Aided Verification*, June 1990.

[SSS+93] D. Schmidt, B. Stiller, T. Suda, A.N. Tantawy, and M. Zitterbart. Language support for flexible application-tailored protocol configuration. In *LCN 93*, 1993.

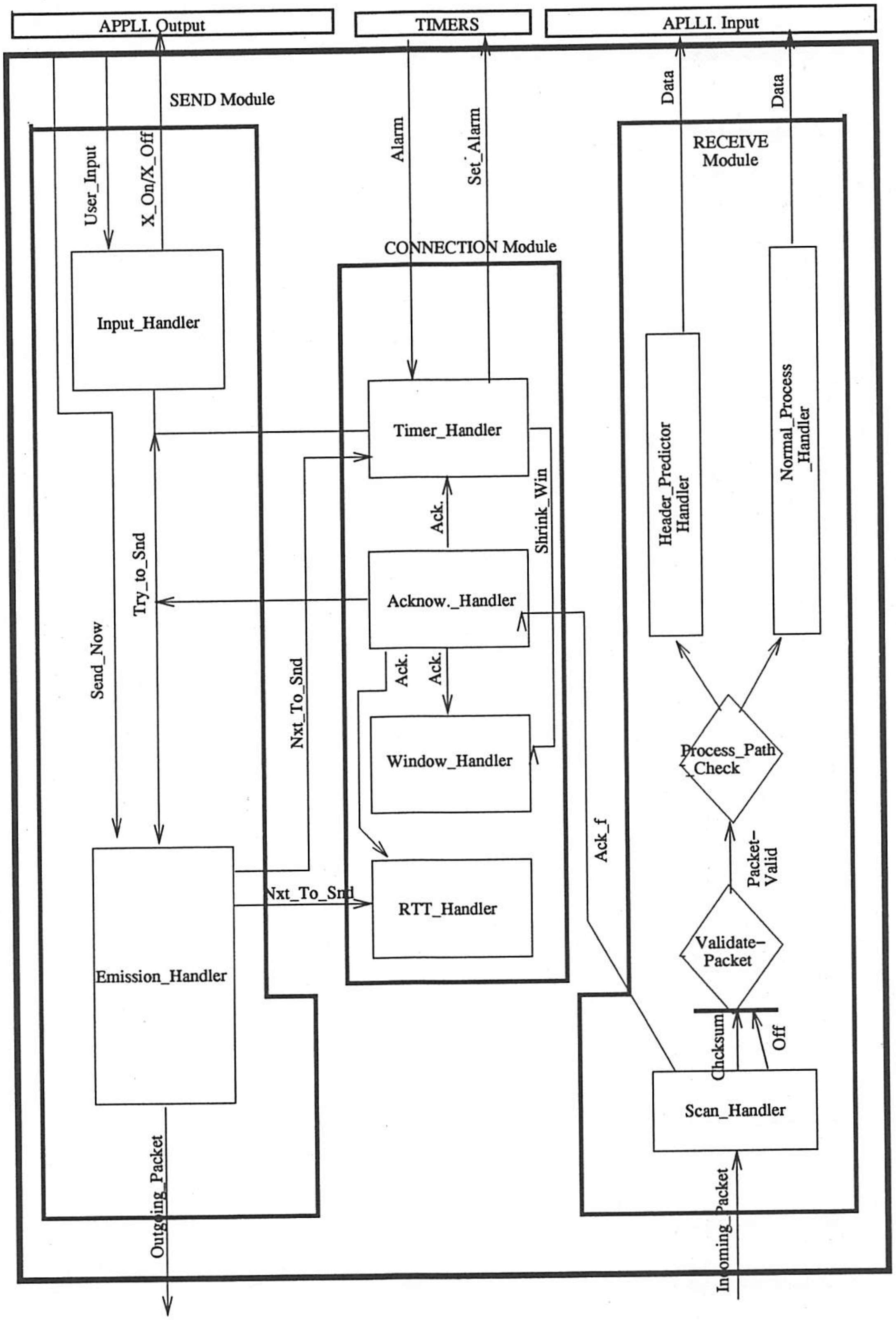


Figure 1: Protocol implementation Overview

# A Framework for the Non-Monolithic Implementation of Protocols in the $x$-kernel

Parag K. Jain, Norman C. Hutchinson, and Samuel T. Chanson*
*Department of Computer Science*
*University of British Columbia*
*Vancouver, B.C., Canada.*
*Email : { jain, hutchinson, chanson }@cs.ubc.ca*

**Abstract**

This paper describes a framework for splitting the implementation of network protocols in the $x$-kernel into a user-level library, which implements the majority of the protocol functionality, and a kernel component, which provides a *fast track* to demultiplex packets into the appropriate recipient address space. The design also provides the user with flexibility to decide which aspects of the protocol implementation should be in user space and which should be in the kernel. We implement an example TCP/UDP-IP-Ethernet protocol stack and report on its performance which is competitive with monolithic implementations. The kernel level demultiplexing of network packets is well structured, modular, and scalable. The cost of demultiplexing is shown to be insensitive to the number of user-level instances of protocols. The performance of the approach on a multiprocessor system shows that non-monolithic protocol implementations are potentially more parallelizable than traditional monolithic ones.

## 1 Introduction

This paper describes a framework for service decomposition of communication protocols in the context of the $x$-kernel. It defines a flexible way to divide protocol functionality between the kernel and user level applications such that the kernel provides a *fast track* to demultiplex the packets received from the network and route them to the appropriate recipient address space. At the same time, it allows the user to decide what additional protocol services would be better implemented in the kernel and move them to the kernel.

Traditionally, network protocols have been implemented as part of the kernel. With the advent of the microkernel architecture, there have been attempts to implement protocols in user space. For example, in the Mach 3.0 environment, network protocols are implemented as trusted user-level servers [3]. More recent efforts, however, have been focused on implementing protocols as user level libraries [8, 21]. In these implementations, a *packet filter(PF)* installed in the kernel demultiplexes the packets to the correct recipient address space. Examples include the CMU/Stanford Packet Filter (CSPF) [11], the BSD Packet Filter (BPF) [9], and the Mach Packet Filter (MPF) [22]. Advancements in packet filter technology have improved both the functionality and performance of packet filters to the point that user level protocol implementations can now be constructed that achieve better performance than previous kernel-level implementations [8].

While recent advances in packet filter technology have addressed their performance problems, they have not provided any additional structure or architectural support for protocol implementors.

*On leave at the Hong Kong University of Science and Technology.

What was once an unstructured kernel level protocol implementation becomes, with the addition of a packet filter, an unstructured user level protocol implementation. This paper takes a different approach to the problem of kernel level demultiplexing of network packets. It is based on the $x$-kernel, a widely used protocol framework for single address space protocol implementations, extending it to allow the implementation of each protocol to be separated into a kernel resident component and a user library component. The architecture requires that the kernel component provide at least the efficient demultiplexing of incoming network packets. Additional functionality can be incorporated in the kernel component as necessary on a protocol by protocol basis. Our framework provides the following functionality:

- Protocols are implemented as user-level libraries.
- There is a fast track in the kernel to route packets to the appropriate user space. The overhead remains constant as the number of protocols in user space increases.
- The framework deals with idiosyncrasies of different protocols in a flexible way. The user can decide what protocol functionality to configure in user space and what in the kernel. However, the design goal would be to place as much functionality as possible in the user's address space.
- In the extreme, it is possible to configure the same protocol in the kernel for one application, and in user space for another application.

## 2 Motivation and Previous work

Two previous efforts to split the implementation of communication protocols between the kernel and user space are described in [8] and [21]. They point out the main advantages and motivation for the user level implementation of protocols. These advantages can be summarized as follows:

- The protocol code is decoupled from the kernel and hence can be easily modified and customized on an application specific basis.
- During the protocol development phase, it becomes easier to debug and experiment with new protocols.

We observe the following additional advantages:

- Reduced contention because each user has its own protocol stack. This implies that multiple instances of the protocol need not share global state.
- Increased parallelism since user level implementations can exploit the parallelism provided by the new generation of multithreaded environments and parallel machines due to the smaller number of synchronization points in the shared protocol state.
- Performing only demultiplexing operation in the kernel integrates the demultiplexing operation of different protocol layers into a single "one-shot" operation, while maintaining compatibility with existing protocols. This is in agreement with the recent literature [2, 20] which argues that performing multiplexing/demultiplexing at multiple layers in the protocol stack significantly reduces throughput.

Previous user-level protocol implementations [8, 11] are based on having a packet filter in the kernel which determines, for each packet, which user level address space should receive the packet. We believe that packet filter technology [9, 11, 22] has a number of limitations:

- Packet filters are based on interpreted languages. The interpreted-language approach provides excellent support for the run-time installation of the packet filter. However, demultiplexing of network packets (i.e. packet filtering) is also interpreted. The experience of the programming language community argues that interpreted languages are often an order of magnitude slower than compiled ones [16].

- Interpretation of some headers fields is done twice: once during packet filtering and once again in the application to locate the correct session (e.g. protocol control block).
- Packet filters suffer from a lack of modularity in that each packet filter must mention every protocol in the path between the network device and the user. For example, while the transport protocol (UDP or TCP) installs the packet filter, that packet filter depends on the network protocol that will eventually deliver the data, thus losing the modularity advantage of IP which is supposed to shield high level protocols from network protocol details.
- Packet filters are not general enough to deal elegantly with the idiosyncrasies of current communication protocols. As an example, consider the IP [13] protocol. It is difficult for a packet filter to handle the reassembly of IP fragments, IP forwarding, and IP tunneling.[1] Of all the packet filter implementations reported in the literature, only the Mach Packet Filter [22] provides a solution to IP reassembly, and we believe that not even packet filter proponents can argue with conviction that the solution is elegant. With the packet filter approach, IP forwarding and IP tunneling are currently supported by running user-level servers which incur additional performance costs in the form of extra context switches and additional packet data copying. We believe that these operations can be performed much more efficiently in the kernel and that a general protocol architecture should support such an implementation.
- It is difficult for a packet filter to act on protocol specific real-time information that may be either encoded in network packets or negotiated for particular network connections.

Proponents of packet filter technology argue that the performance problems of packet filters can be addressed by good engineering [22], and that making the packet filter do more (like handling IP fragments) is as easy as adding a few new instructions [22]. In many ways, the recent changes to the packet filter amount to a way of creating a kernel resident protocol entity complete with support for multiple connections and even connection specific state. Our solution takes these trends to their logical conclusion: what is really wanted is the ability to embed in the kernel a small fragment of the functionality of the communication protocol. This paper describes our efforts to do exactly that.

**Organization of this Paper**

The rest of this paper is organized as follows: section 3 presents an overview of the $x$-kernel which is essential in understanding the rest of this paper. Section 4 describes the design and architecture of our protocol framework and section 5 describes the implementation of an example TCP/UDP-IP-Eth protocol stack. Section 6 analyzes the performance of the example protocol stack. Finally section 7 offers some conclusions.

## 3 The $x$-kernel Overview

The $x$-kernel [6, 7] is a protocol implementation architecture that defines protocol independent abstractions which facilitate the efficient implementation of communication protocols. It provides three primitive communication objects: *protocols*, *sessions*, and *messages*. Each protocol object corresponds to a communication protocol such as IP [13], TCP [15], or UDP [12]. A session object represents the local state of a network connection within a protocol and is dynamically created by the protocol object. Messages contain the user level data and protocol headers and visit a sequence of protocol and session objects as they move through the $x$-kernel.

Each protocol implements a set of methods which provide services to both upper layer and lower layer protocols via a set of generic function calls provided by the *Uniform Protocol Interface(UPI)*. With the UPI interface, protocols do not need to be aware of the details of protocols above or below them in the protocol graph. By using a dynamic *protocol graph*, the binding of protocols into

[1]Tunneling implies that a complex network with its own protocols is treated like any other hardware delivery system.

hierarchies is deferred until link time. The protocol graph represents the static relationship between protocols. In the $x$-kernel, user processes are also treated as protocols. Hence, the $x$-kernel provides *anchor* protocols at the top and bottom of the protocol hierarchy which defines the application interface and device driver interface respectively. The anchor protocols allow easy integration of the $x$-kernel into any host environment.

The $x$-kernel provides a set of support tools [5, 7] to implement protocols efficiently and quickly. These tools consist of a set of routines that are commonly used to implement protocols. The major tools include a map manager, message manager, process manager and event manager.

## 3.1 Operations on Protocol Objects

The two main functions of protocol objects are management of session objects, and demultiplexing received messages to the appropriate session objects.

The protocol object supports three operations to create session objects: *open, open_enable,* and *open_done.* A high level protocol creates a session object in a lower layer protocol by invoking a lower protocol's *open* routine. Alternatively, a high level protocol can pass a capability to a lower level protocol for passive creation of session at a future time by invoking the *open_enable* routine of the lower protocol. The *open_enable* operation creates an *enable* object to store the capability passed by the upper layer protocol. On receipt of a message from the network, the lower layer protocol creates a session and signals the creation of a session by invoking the upper layer protocol's *open_done* routine.

The above three routines take a *participant_list* as one of their arguments. The *participant_list* is a name for some participant in a communication, and consists of a sequence of protocol specific *external ids*[2] such as port numbers, protocol numbers, or type fields used by the protocols to identify message recipients. The mapping between external ids and *internal ids,* which are capabilities for session objects or enable objects, is maintained in the *active* and *passive* maps maintained by each protocol. The maps are implemented as hash tables which cache the last key looked up.

In addition to managing connections, the primary role of the protocol object is to demultiplex incoming data to the appropriate session. This is accomplished by a routine called *demux* which extracts certain fields from the header part of the message (e.g. source and destination port numbers) and constructs an external id. If a binding to a session is found in the active map, then the message is delivered to that session. If no session is found, and searching in the passive map yields an enable object, then a new session is created as described previously. Otherwise, the message is dropped.

## 3.2 Operations on Session Objects

Sessions support two primary operations - *push* and *pop.* The protocol's *push* routine appends the protocol header to the message and passes the message to a lower layer session. The protocol *pop* routine updates the state of the connection represented by the session and passes the message to an upper layer protocol.

## 4 Design Overview

Before describing our design, we shall first explain the terms *kernel, user* and *IPC. Kernel* refers to the operating system kernel or the user level trusted server that implements the kernel component of protocol functionality. By *user,* we mean the user level application that links to the user level protocol libraries. The means of communication between kernel and user is referred to as *IPC.* In the case when the kernel is an operating system kernel, then the IPC will mean a system call or upcall. If it is a trusted server, the IPC will mean inter-process communication.

The main idea is to decompose protocol implementations into two parts: one part in the kernel and the other linked with the application as a user level protocol library. The kernel level part

[2]Also referred to as *key* in this paper.

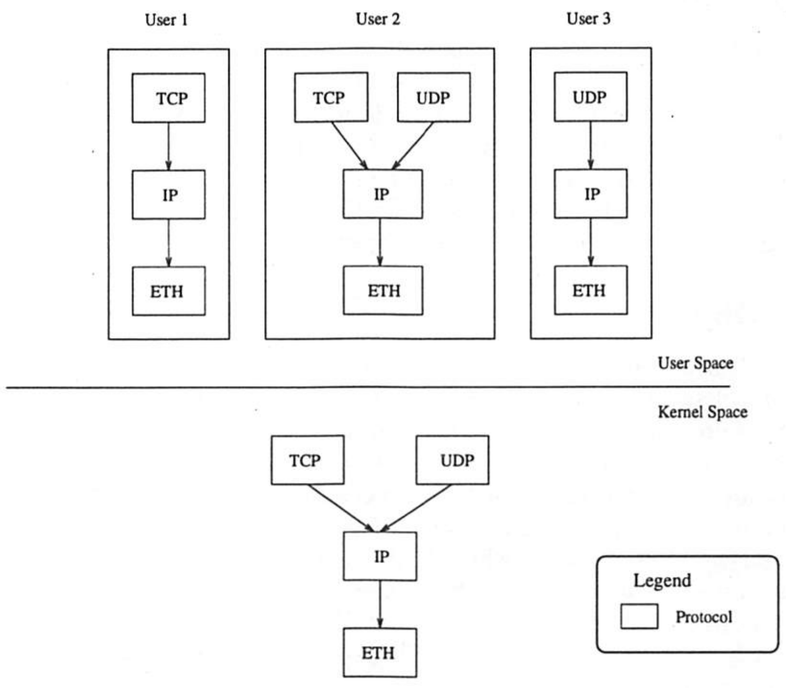


Figure 1: User and Kernel Protocol Graphs

implements, at a minimum, the demultiplexing of packets received by the protocol. In an efficient implementation, the cost of demultiplexing should not increase with the number of user level instances of protocols. In earlier implementations based on the packet filter [11], the cost of demultiplexing increased linearly with the number of connections. The Mach Packet Filter achieves performance that is not sensitive to the number of open connections by collapsing the common parts of different packet filters into a single filter [22]. In our scheme, we exploit the notion of protocol graphs to achieve this. These protocol graphs define the protocol hierarchy as explained in the $x$-kernel overview. In a naive user level protocol implementation the demultiplexing operation will be performed twice – one time in the kernel to locate the recipient application and then again in the application to locate the recipient session (i.e. protocol control block) because, in general, an application may open more than one connection. In an efficient implementation, the demultiplexing in the application can be completely avoided. Our scheme avoids demultiplexing in the user by having the user register the user session identifiers with the kernel as described later in this section.

## 4.1 Protocol Graphs

In our design, each user has its own protocol graph, as does the kernel. The kernel level protocol graph is the *union* of all user level protocol graphs and is called the *Universal Protocol Graph(UPG)*. The user level protocol graph is registered with the kernel when the user application starts. The kernel incorporates this graph into the UPG by the *union* operation. Figure 1 shows the protocol graphs of 3 different users and the corresponding kernel protocol graph. The entire suite of protocols which is available to each user appears in its protocol graph even if some of those protocols are actually implemented in the kernel.

Each protocol's demultiplexing routine is also registered with the kernel. This routine is protocol specific and not user (or protocol instance) specific. This implies that although there are a number of copies of the same protocol at the user level, the kernel will use a single demultiplexing routine for this protocol. This is equivalent to combining different packet filters (e.g. Mach Packet Filter [22]).

## 4.2 Protocol Objects

The protocol objects reside in the kernel as well as in the user space. However, the kernel level protocol objects in most cases are skeletons of user level protocol objects and implement only that portion of the protocol functionality that the user decides to move to the kernel. We call them *skeleton* protocols. The minimum functionality of a protocol that resides in the kernel is demultiplexing. However, a user may decide to move more functionality to the kernel, and in the extreme, our design allows a user to move the complete protocol stack to the kernel. This flexibility is achieved by adding new data structures to the protocol and session objects.

## 4.3 Session Objects

In a user level implementation of protocols, sessions need to reside in the application because they store all the protocol state information of the application's connections. Since demultiplexing of incoming packets is done in the kernel, we need to have sessions in the kernel as well. The kernel level sessions are proxies for their user level counterparts. We call these sessions *shadow sessions.*

In the *x*-kernel, an *open* call leads to the creation of a session at each protocol layer. These sessions are linked by pointers and form a session stack. Every user level open call is registered with the kernel in a *RegisterOpen* IPC message which leads to the creation of a shadow session stack in the kernel. The user level also sends the user IPC port and user level *session id list* in the IPC message. This information is stored in the shadow session at the user-kernel boundary and is passed to the user along with each received network packet. This avoids extra demultiplexing in the user space to locate the appropriate session.

User level passive opens (e.g. a server waiting on a port for connections) are also registered with the kernel by using a *RegisterOpenEnable* IPC message. This enables the kernel to create a shadow session stack when a packet is received for such a server.

## 4.4 Demultiplexing in the kernel

Demultiplexing in the kernel proceeds in the traditional *x*-kernel way. However, there are no *pop* routines in the kernel. The UPI library's *xPop* routine is modified so that instead of calling the protocol specific *pop* routine, it registers the protocol specific header information for later playback at the user level.

In the kernel, at each protocol layer, the protocol specific *demux* routine is invoked, the header information is stored in an IPC buffer, and the message is passed to the upper layer protocol. At the user-kernel boundary layer protocol (e.g. TCP, UDP, etc), the demultiplexing determines the identity of the destination user level application for the current message. The session returned by the demultiplexing routine at this protocol layer contains the user level IPC port number and the user session id list. The message is then transferred to the destination user space along with the user session id list and the bookkeeping information recorded by the kernel *xPop* routine in a *ReceivePacket* IPC call.

At each protocol layer, the protocol specific demultiplexing routine is invoked. There is no header processing in between the demultiplexing routines as in a traditional implementation. This is equivalent to combining the demultiplexing operation of multiple protocol layers into a single "one-shot" operation. This also makes it possible to take an Integrated Layer Processing approach to the data manipulation [2, 20].

## 4.5 Packet Processing at User Level:

The user level receives the incoming packets from the kernel along with bookkeeping information and the user session id list. The connection specific protocol state is updated by invoking a series of *pop* routines, resulting in passing the user data to the application. A user may passively wait for connections. In this case, when the first packet is received on a connection, the user level sessions are created. The user level session stack thus created is then registered with the kernel.

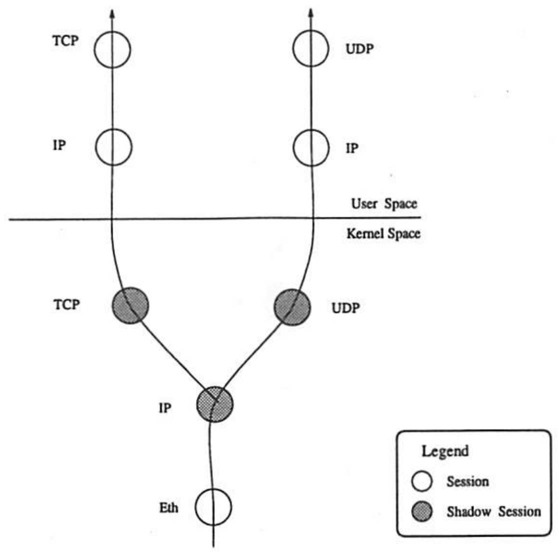


Figure 2: Incoming message paths

Sending packets out is straightforward. For outgoing messages, protocol headers are appended to the message at each protocol layer by invoking a series of *push* routines. At the user-kernel boundary protocol, the message is transferred to the kernel level protocol in a *SendPacket* IPC message. The kernel level protocol to which the message is sent is the one that logically sits just below the bottom user level protocol in the protocol graph hierarchy. This information is readily available from the user level protocol graph. The kernel is free to make any checks required for security reasons in the header portion of the outgoing message.

Figure 2 and 3 depict the receive and send paths for incoming and outgoing messages respectively. In the figures, the TCP, UDP, and IP protocols are configured in the user space and have corresponding skeleton protocols in the kernel. The Ethernet [10] protocol is configured in the kernel, but to improve the performance of outgoing packets, we configure a skeleton Ethernet protocol in the user space. The figures also illustrate that kernel level sessions that are not at the user-kernel boundary (e.g. IP, Ethernet) may be shared by more than one user level protocol belonging to the same or different applications.

## 4.6 Shared state

One of the problems that must be addressed in any protocol implementation that admits multiple implementations of protocols is the problem of shared protocol state. Examples of such state include IP routing information and ARP tables which map between IP and network addresses. Our framework supports a simple but general protocol independent mechanism to manage such information assuming that one address space (typically the kernel) maintains the master copy of such information and other address spaces (typically the users) wish to maintain caches of subsets of the information. This facility amounts to an IPC broadcast between the kernel implementation of a protocol and all of the user implementation of that same protocol whenever the shared state is changed. This facility is also required to handle IP broadcast and multicast packets.

## 4.7 More Details of Skeleton Protocols

The skeleton protocols need to manage kernel sessions for demultiplexing incoming network packets. It is possible to provide a set of generic routines such as *open/open_enable/close/open_disable* to perform these connection management functions in the kernel but the *demux* routine is protocol

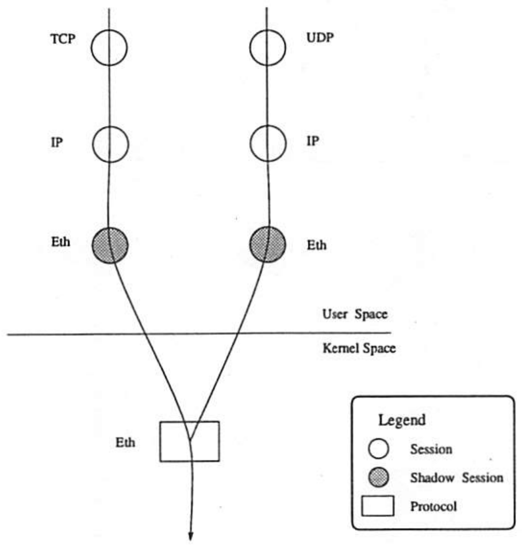


Figure 3: Outgoing message paths

specific and therefore, must be implemented by the protocol writer.[3] In the most common case, this is all that is necessary to construct the skeleton protocol in the kernel. If the protocol writer decides to move additional functionality of the protocol to the kernel, he/she must provide protocol specific routines as necessary that supersede the generic routines.

To give insight into how the skeleton protocols function, we now describe how the generic *open/open_enable* functions proceeds as an example. The *close* and *open_disable* routines are similar. These routines take a participant list as one of their arguments. The participant list is supplied by the user in the *RegisterOpen/RegisterOpenEnable* calls and is created during the *open/open_enable* call in the user level protocols. As the user level active open (i.e *open*) propagates down the user protocol stack, each protocol records the key it used to identify the newly created session object. When the user level *open* call completes, the top level protocol has the list of the keys each protocol used for identifying its sessions. It passes this list to the kernel to initiate the creation of a shadow session stack in the kernel. The kernel level protocols treat this list as a stack – each protocol pops a key from the stack, creates a shadow session, and passes the rest of the stack to the next lower layer protocol to initiate creation of shadow sessions at lower layers.[4] A passive open (i.e *open_enable*) is similar to an active open except that a passive open does not propagate down the protocol stack.

## 4.8 Protocol Configuration

Each protocol, session, and enable object contains a new bit called the *ukBoundaryBit* which specifies whether the object resides at the user-kernel boundary. The user level protocols which are at the bottom of the user level protocol graphs will have this bit set indicating there is no lower level user space protocol, as will the kernel level shadow sessions at the top of the kernel protocol graph. This permits the kernel to know if the received packet needs to be transferred to the destination application. The shadow session's *state* data structure stores the user level port and user *session id list*.

[3]It is difficult to provide a generic *demux* routine because protocol header length is not always constant (e.g. TCP and IP).

[4]Some protocols such TCP and UDP use the lower layer protocol session (e.g. the IP session) to form the keys for demultiplexing a received packet into a session. We define a special symbol *USE_LOWER_SESSION* as the key to tell the kernel level protocol to use the lower layer session as the key.

Currently, protocol configuration is done statically. The kernel resident active and passive maps for each protocol are created at the system startup time. The key size information required to create these maps is stored in the protocol graph and hence is readily available. Our architecture would easily accommodate dynamic protocol configuration, but our environment does not provide a mechanism for the dynamic addition of code to the kernel (such as loadable device drivers).

## 5 An Example Implementation

We have implemented the split protocol stack for UDP [12]/TCP[15], IP [13], ARP [17], ICMP [14], and Ethernet[10] protocols. UDP, TCP, and most of IP, ARP, ICMP are in the user library while the remaining parts of IP, ARP, and Ethernet sit in the kernel. Some IP functions are implemented in the kernel as will be explained later. ARP is implemented at user level as well as in the kernel. The kernel level ARP is the master and periodically broadcasts the ARP table changes to the user level instances of ARP as described earlier.

TCP and UDP port assignment must be centralized for correct functioning. In our scheme, the only central point is the kernel, so port assignment is done by the kernel. To improve the performance of outgoing packets, we configure a skeleton Ethernet protocol in the user space library. This caches Ethernet header information from the real Ethernet driver sitting in the kernel.

When a user level application wants to set up a TCP connection to a remote machine, it calls *tcp_open* with the appropriate set of participants. The *tcp_open* routine creates a TCP session and calls open on the IP protocol. IP creates a new session or returns an already existing session. IP also links to the appropriate network interface (in our case Ethernet). This completes the creation of a user level session stack. The user session id list is then registered with the kernel by sending a *RegisterOpen* IPC request to the kernel which creates the shadow session stack in the kernel. The kernel stores the user level session id list in the newly created shadow TCP session's *state* data structure. Next, *tcp_open* tries to establish a connection with the remote machine. If the connection is successfully established, *tcp_open* returns success. Otherwise it destroys the user level session stack that has just been created and sends a *DeregisterOpen* IPC request to the kernel to destroy the kernel level shadow session stack.

On the remote machine, the server waits passively for new connections after having registered a *RegisterOpenEnable* with the kernel. When the connection message is received on the remote machine, the kernel demultiplexes the packet and ultimately creates the shadow session stack in the kernel. It then passes the packet to the server via the *ReceivePacket* IPC message. The protocol library in the server creates the user level session stack as the message flows up the protocol stack. At the TCP level, after the TCP session is created, the user level session id list is registered with the kernel in the IPC reply message to the *ReceivePacket* IPC call. The kernel stores the user session id list in the corresponding TCP shadow session. This completes the creation of the user level session stack as well as the kernel level shadow session stack on the remote machine. Hereafter, packets can be efficiently exchanged between the two machines with demultiplexing done only in the kernel, and protocol specific connection state processing only in user space.

### IP Services Decomposition

The IP protocol implementation has been split between the kernel and user space such that the services that are difficult to handle in user space are implemented in the kernel. These services are IP reassembly of fragments, IP routing table management, IP packet forwarding, IP broadcasts, and IP multicast packet processing. The kernel thus maintains enough of the IP protocol context so that it can handle these services. IP Routing tables are maintained in the kernel and periodically broadcast to the user level instances of IP. The kernel maintains maps for collecting IP fragments and when a message is reassembled, it is passed to the upper layer protocol (in our case UDP). The upper level protocol transfers the complete message to the appropriate user space. This implies that the user level IP never has to deal with reassembly issues. Similarly, the kernel maintains the IP forward map which stores the session corresponding to packets that need to be forwarded.

## 6 Performance

In this section, we report on the performance of the sample application level protocol stack (TCP/UDP-IP-Ethernet) implemented in our framework and compare it with the performance of the monolithic implementation. We first describe the hardware and software platform for our experiments and the benchmark tools used. We then report on the round trip latency, incremental cost per round trip, latency breakdown for receive and send paths, the kernel level demultiplexing cost, TCP throughput, and connection setup cost in the context of the sample protocol stack. Finally, we present the effect of parallel processing on the round trip latency by using more than one processor.

### 6.1 Hardware/Software Platform

The hardware platform for our experiments is the Motorola MVME188 Hypermodule, a 25 MHz quad-processor 88100-based shared memory machine [4]. A light weight, multithreaded kernel called *Raven* [18, 19] has been developed in the Computer Science department at UBC and is the microkernel running on the bare hardware. The protocols have been implemented in the context of a parallel $x$-kernel: the $x$-kernel [6, 7] has been ported as a user level server in this environment and has been parallelized to take full advantage of parallel processing. Our approach to the parallelization of the $x$-kernel is similar to the one described in [1]. We use the one processor/thread per packet model for the processing of incoming and outgoing packets. The performance measurements are made by connecting two Motorola MVME188 Hypermodules through a 10 Mb/sec Ethernet network in light network traffic conditions.

**IPC Design**

We use Raven's light-weight synchronous IPC library which associates a shared buffer with each communication port. In order to achieve pairwise shared buffers, each application initializes by going through a registration process with the kernel in which each side creates a communication port and an appropriate number of communication threads and sends the port identifier to the other side. The application stores the kernel port identifier in the protocol objects at the bottom of its protocol graph and directs all but its first request to the newly created kernel port. Likewise, the kernel stores the port identifier of the application in the shadow session, and enable objects that are created on behalf of the application at the user-kernel boundary.

**Benchmark Tools**

We have analyzed the performance of our implementation in two ways.

- Round trip times and timing of the various components of the implementation have been taken using an on-board Z8536 Counter/Timer configured as a 32-bit timer with microsecond resolution.
- Instructions have been counted by using the instruction tracing facility of a hardware simulator.

Before giving the performance numbers, we briefly describe the difference in the various implementations. We describe three versions: *Server-Server*, *User-User Server-Stack* and *User-User Partitioned-Stack*. *Server-Server* implementation is the traditional $x$-kernel running as a user level server. There is no boundary crossing between the user and the $x$-kernel server. In *User-User Server-Stack* case there is a boundary crossing between the $x$-kernel server and applications, however, all the protocol processing is done in the $x$-kernel server. The *User-User Partitioned-Stack* version is our new implementation of protocols as a user level library.

### 6.2 User-to-User Latency

We measured the round trip time of various user level protocols and compared it with the corresponding server-stack implementation. The latency test is a simple ping-pong test between two user

| *Protocol Stack* | *Round-Trip Time(ms)* | | | | |
|---|---|---|---|---|---|
| | *User Data Size(bytes)* | | | | |
| | 1 | 100 | 500 | 1000 | 1400 |
| **Server-Server** | | | | | |
| ETH | 0.88 | 1.01 | 2.09 | 3.44 | 4.51 |
| IP-ETH | 1.04 | 1.20 | 2.27 | 3.64 | 4.71 |
| UDP-IP-ETH | 1.21 | 1.42 | 2.49 | 3.80 | 4.87 |
| TCP-IP-ETH | 1.74 | 2.17 | 3.63 | 5.47 | 6.92 |
| **User-User Server-Stack** | | | | | |
| ETH | 1.86 | 1.94 | 3.12 | 4.56 | 5.77 |
| IP-ETH | 2.06 | 2.27 | 3.44 | 4.96 | 6.25 |
| UDP-IP-ETH | 2.26 | 2.54 | 3.70 | 5.18 | 6.43 |
| TCP-IP-ETH | 2.89 | 3.46 | 5.01 | 7.00 | 8.64 |
| **User-User Partitioned-Stack** | | | | | |
| ETH | 1.88 | 1.98 | 3.13 | 4.51 | 5.65 |
| IP-ETH | 2.17 | 2.35 | 3.47 | 4.88 | 6.08 |
| UDP-IP-ETH | 2.39 | 2.63 | 3.78 | 5.17 | 6.31 |
| TCP-IP-ETH | 3.02 | 3.51 | 5.05 | 6.93 | 8.47 |

Table 1: User-to-User Latency

| | *Incremental Costs(ms)* | | |
|---|---|---|---|
| *Protocol* | **Server-Server** | **User-User Server-Stack** | **User-User Partitioned-Stack** |
| IP | 0.16 | 0.20 | 0.29 |
| UDP | 0.17 | 0.20 | 0.22 |
| TCP | 0.70 | 0.83 | 0.85 |

Table 2: Incremental Costs of Various Protocols per Round Trip (1 byte user data)

level applications called client and server. The client sends data to the server and the server sends the same amount of data back. The average round-trip time is measured over 10000 such transactions. The figures reported are measured for single processor case. The UDP protocol processing time does not include UDP checksum overhead. Table 1 gives the latency of the various schemes. The latency difference between *Server-Server* and *User-User Server-Stack* is due to the overhead of moving the data between the two address spaces involved. The difference between *User-User Server-Stack* and *User-User Partitioned-Stack* gives the overhead of our scheme. The latency of the Ethernet stack is approximately the same in the server-stack and the Partitioned-Stack cases because their implementations are very similar. We observe that the latency of the Partitioned-Stack is smaller than that of the Server-Stack for user data size larger than 1000 bytes. This, we suspect, is due to the difference in the cache performance for the two implementations.

The following list describes in detail the overhead of our implementation:

- Overhead of recording bookkeeping information at each skeleton protocol layer. The information stored consists of a protocol identifier and the length of the protocol header.
- Longer IPC messages are exchanged because the bookkeeping information must be included in the IPC messages.
- Constructing and interpreting the different types of IPC messages to pass them to the appropriate protocol. This applies to both the kernel as well as the user.

- Preparing the $x$-kernel message object from the IPC message received from the kernel to invoke the series of *pops* routines in the user space.

All of the above overheads are small, requiring only a few instructions, and are independent of packet length.

Table 2 lists the incremental cost of a round trip for each protocol for each implementation. The incremental cost is calculated by subtracting the measured round trip latency for pairs of appropriate protocol stacks that transfer 1 byte user data. For example, TCP latency is computed by subtracting latency for IP-ETH stack from the latency for TCP-IP-ETH stack. In the Partitioned-Stack case, the IP protocol has a much higher incremental cost. This is mainly due to the overhead of preparing an $x$-kernel message object from the IPC message. This cost is incurred only by protocols that are at the user-kernel boundary. The incremental cost of TCP and UDP in the Partitioned-Stack case is 20 microseconds higher than that in server-stack case. This can be attributed to the small overhead of bookkeeping information and the larger IPC messages that are exchanged. We also observe that the incremental cost of each protocol in the server-stack case is higher than that of the corresponding protocol in the Server-Server case. This, we suspect, is due to unfavorable cache performance since two address spaces are involved in the server-stack case.

### 6.2.1 Latency Breakdown

Table 3 gives the time spent in various protocol layers in the kernel and the user space for both send and receive paths. Tables 3 and 4 also list the overheads involved in recording and playback of bookkeeping information.[5] *IP PopTcb* and *TCP/UDP PopTcb* reflect the cost of recording bookkeeping information while *IP DemuxTcb* and *TCP/UDP DemuxTcb* reflect the cost of the playback of bookkeeping information. As can be seen, the number of instructions executed for these functions is extremely small. The user-kernel boundary protocols spend more time in recording the bookkeeping information because they also store the user level session list in the IPC message. The *Other Overheads* reflect the cost of function calls and $x$-kernel UPI library function calls. The *User IP Preprocessing* reflects the cost of converting an IPC message back into an $x$-kernel message object. This component is the largest single overhead of our Partitioned-Stack implementation over the server-stack one.

In the send path, the cost of the Ethernet layer increases dramatically with the size of the user data because the message is copied to a contiguous buffer before making each IPC call. This extra copy could be avoided by modifying the IPC library. The IPC cost from user to kernel is more than the IPC cost from the kernel to the user. Again, we suspect this is the due to the poor cache performance in the first case.

### 6.2.2 Kernel Level Demultiplexing

The kernel demultiplexing takes 61 to 98 microseconds. The large variation in the demultiplexing cost is related to cache performance.[6] Therefore, we give a breakup of the number of Motorola 88100 instructions executed in each kernel level protocol layer's demultiplexing routine and the total number of instructions taken for demultiplexing in the kernel. Table 4 lists the number of instructions taken by various components of the kernel demultiplexing and user level processing. The demultiplexing cost for the TCP-IP-ETH stack is the same as that of the UDP-IP-ETH stack and they take 687 and 686 instructions respectively. Given this low cost of kernel level demultiplexing, we could perform the complete demultiplexing function in the interrupt handler. This would avoid one extra copy of the data as we would be able to copy each packet directly to the IPC buffer shared between the recipient address space and the kernel.

Figure 4 shows the cost of kernel level demultiplexing as the number of open TCP connections is varied from 1 to 100. The solid line indicates the cost of demultiplexing when the data is sent

[5]It is possible to get rid of the bookkeeping overhead in a customized implementation. In that case, the number of instructions taken for kernel demultiplexing is further reduced.

[6]The cache size on our machine is 16KB instruction and 16 KB data cache per processor.

| *Protocol Component* | *TCP(microseconds)* |  | *UDP(microseconds)* |  |
|---|---|---|---|---|
|  | *Length(Bytes)* |  | *Length(Bytes)* |  |
|  | 1 | 1400 | 1 | 1400 |
| **Send Path** |  |  |  |  |
| TCPtest/UDPtest | 24 | 26 | 15 | 15 |
| TCP/UDP layer | 179 | 561 | 45 | 47 |
| IP layer | 30 | 30 | 30 | 30 |
| Other Overheads | 6 | 6 | 6 | 6 |
| Ethernet Layer | 70 | 235 | 65 | 225 |
| IPC to Kernel | 294 | 391 | 285 | 355 |
| *Ethernet Driver* | 66 | 275 | 62 | 270 |
| **Total Send Path** | 669 | 1524 | 508 | 948 |
| **Receive Path** |  |  |  |  |
| Interrupt Dispatch and Handling | 96 | 89 | 96 | 91 |
| Ethernet read | 67 | 361 | 63 | 351 |
| *Ethernet demultiplexing* | 17 | 17 | 17 | 17 |
| *IP demultiplexing* | 21 | 21 | 21 | 21 |
| *IP PopTcb* | 4 | 4 | 4 | 4 |
| *TCP/UDP demultiplexing* | 18 | 17 | 18 | 18 |
| *TCP/UDP PopTcb* | 8 | 8 | 8 | 8 |
| *Other Overheads* | 13 | 13 | 13 | 13 |
| IPC to User | 246 | 309 | 238 | 306 |
| User IP Preprocessing | 46 | 46 | 46 | 47 |
| User IP layer | 47 | 45 | 47 | 47 |
| User TCP/UDP layer | 186 | 565 | 46 | 47 |
| TCPtest/UDPtest | 6 | 6 | 6 | 6 |
| Other Overheads | 16 | 16 | 16 | 16 |
| **Total Receive Path** | 791 | 1517 | 639 | 992 |
| Network Transit Time | 50 | 1215 | 50 | 1215 |
| **Total** | 1510 | 4256 | 1197 | 3155 |

Table 3: Latency Breakdown — User-User Partitioned-Stack case

| *Protocol Component* | *TCP* | *UDP* |
|---|---|---|
|  | *# Instructions* | *# Instructions* |
| *Ethernet demultiplexing* | 154 | 154 |
| *IP demultiplexing* | 230 | 230 |
| *IP PopTcb* | 13 | 13 |
| *TCP/UDP demultiplexing* | 172 | 171 |
| *TCP/UDP PopTcb* | 33 | 33 |
| *Other Overheads* | 85 | 85 |
| **Total Kernel Demultiplexing** | 687 | 686 |
| User IP DemuxTcb | 7 | 7 |
| User TCP/UDP DemuxTcb | 7 | 7 |

Table 4: Instruction Counts — User-User Partitioned-Stack case

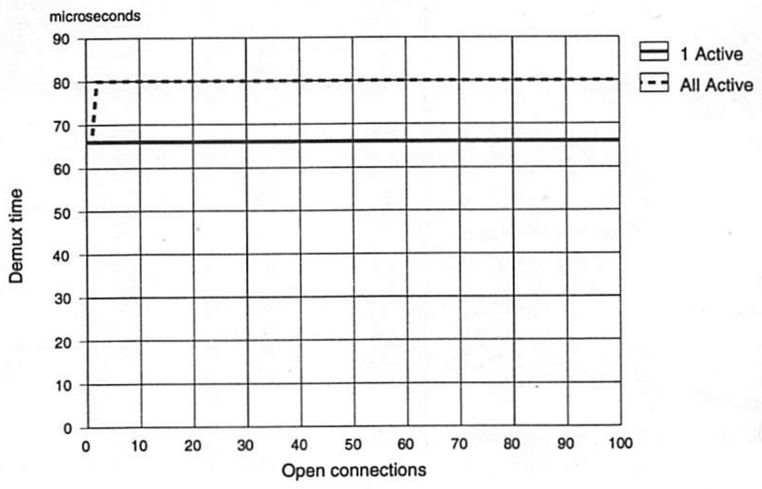


Figure 4: Demultiplexing Cost as a Function of Number of Open Connections

| *Protocol Stack* | *Throughput(Mb/sec)* |
|---|---|
| **Server-Server** | 6.15 |
| **User-User Server-Stack** | 4.74 |
| **User-User Partitioned-Stack** | 4.30 |

Table 5: TCP Throughput

to only one connection. This is the best case demultiplexing scenario and costs 66 microseconds. The session entry in this case, is always found in the cache of the TCP active map. The dotted line depicts the demultiplexing cost in the worst case scenario in that data message is sent to each open connection in a round robin fashion. In this case, the session entry is guaranteed not to be in the cache of the TCP map and a search in the map has to be carried out. This results in an extra 14 microseconds of overhead. Another important point to note is that the demultiplexing cost is totally insensitive to the number of open connections. This is similar to the result achieved in traditional kernel based protocol implementations and Mach Packet Filter [22] based implementation [8].

## 6.3 User-to-User Throughput

Table 5 lists the throughput in Mb/sec for TCP for various implementations. TCP throughput is measured over the TCP-IP-ETH protocol stack by measuring the time taken to send 1 megabytes from the client to a server. TCP uses a 4096 bytes window and fragments the data into 1460 byte data messages. The throughput differences among the three implementations are as expected. In the Server-Server implementation, there is no boundary crossing from the server to the application and hence, its performance is the best. In the Server-Stack implementation there are two boundary crossings on each side, per packet, and a corresponding performance reduction is observed. In the Partitioned-Stack implementation, extra overhead over the Server-Stack implementation is incurred because of longer IPC messages that are exchanged, converting each IPC message to the $x$-kernel message object, and recording and playback of bookkeeping information. This overhead results in a 9% slowdown over the Server-Stack implementation. In the Partitioned-Stack implementation,

| *Protocol Stack* | *TCP Connection Setup time(ms)* |
|---|---|
| **Server-Server** | 5.81 |
| **User-User Server-Stack** | 7.42 |
| **User-User Partitioned-Stack** | 11.01 |

Table 6: Connection Setup Cost

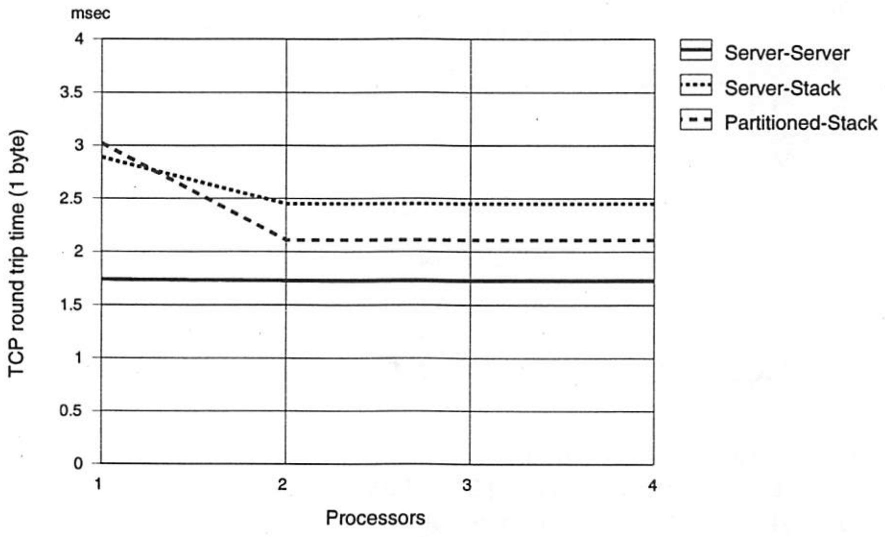


Figure 5: TCP Latency as a Function of Number of Processors

there is one extra data copy on the send path over the Server-Stack implementation which could be eliminated by changing the IPC library interface. This elimination would result in performance closer to that of the Server-Stack implementation.

## 6.4 TCP Connection Setup Cost

Table 6 indicates the relative connection setup time of the three implementations. The connection setup cost is an important measure of performance for applications that periodically connect to a peer entity, send a small amount of data, and close the connection. The connection setup cost for the Server-Stack implementation incurs IPC overhead over the Server-Server implementation. The 3.59 milliseconds extra for the connection setup cost for Partitioned-Stack implementation is incurred over the Server-Stack implementation because of the following overheads:

- Extra IPC messages exchanged to register the user level session id list with the kernel.
- Creation of a shadow session stack in the kernel.

## 6.5 Performance on Multiprocessors

Figures 5 and 6 show the effect of active multiprocessors on the TCP and UDP round trip latencies respectively for all three implementations. The interesting point to note is that the latency of the Partitioned-Stack implementation is smaller than the Server-Stack latency for more than one active

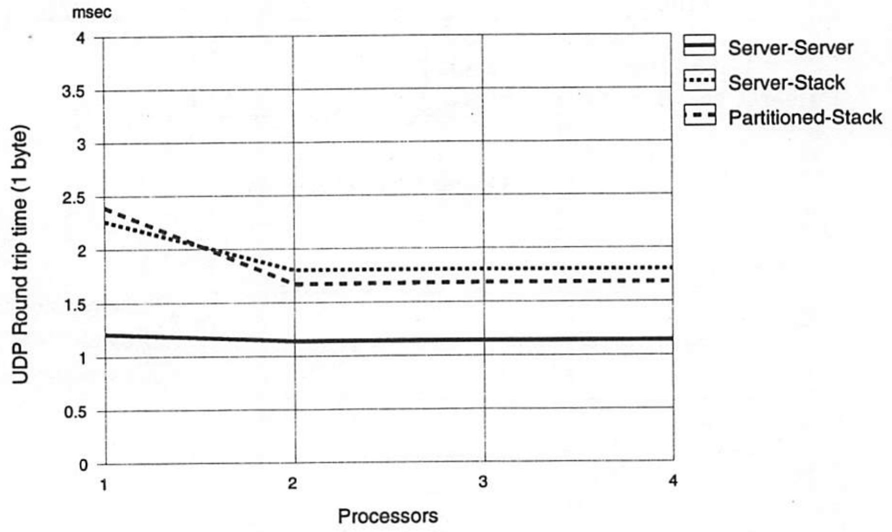


Figure 6: UDP Latency as a Function of Number of Processors

processor. This shows that the partitioned stack implementation is potentially more parallelizable than the Server-Stack or Server-Server implementations. This is an important result because for high-speed networking we believe that shared memory multiprocessor machines will be the most successful. Another interesting point to note is that the latencies of TCP and UDP drop when we go from a single active processor to two active processors and remain constant as the number of active processors is further increased. This is because there is not enough concurrency in the latency tests which are of a stop and wait nature.

## 7 Conclusions

We have described a protocol independent framework for implementing network protocols as user level libraries. This framework provides a flexible way to decompose protocol services between a trusted kernel address space and user address spaces and allows the protocol implementor to decide how the services should be decomposed. The primary functionality that must be included in the kernel portion of the protocol implementation is the efficient demultiplexing of incoming data to the appropriate recipient address space. Additional functionality, like IP reassembly, routing table management, etc., may be added to the kernel implementation when it is considered worth while. We characterize our approach as the logical conclusion of current trends in packet filter technology as more and more support for the idiosyncrasies of particular protocols make their way into the instruction set of packet filters. The performance of our framework was demonstrated through a TCP/UDP-IP-Ethernet protocol stack implementation which is shown to be competitive with a monolithic implementation. The kernel level demultiplexing is shown to be unaffected by the number of open connections. We have also shown that non-monolithic protocol implementations are potentially more parallelizable than the traditional monolithic implementations. This result is important for future high-speed network environments.

## Acknowledgements

We would like to thank Donald Acton, Daniel Hagimont, Jeffrey Mogul, and the anonymous referees for their valuable feedback on early drafts of this paper which greatly improved its quality.

## References

[1] Mats Bjorkman and Per Gunningberg. Locking effects in multiprocessor implementations of protocols. In *Proceedings of ACM SIGCOMM'93 Conference on Communications, Architecture and Protocols*, September 1993.

[2] David C. Feldmeier. Multiplexing issues in communication systems design. In *Proceedings of ACM SIGCOMM'90 Conference on Communications, Architecture and Protocols*, pages 209–219, September 1990.

[3] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *Proceedings of the 1990 Summer USENIX Conference*, pages 87–95, June 1990.

[4] Motorola Computer Group. *MVME188 VMEmodule RISC Microcomputer User's Manual*. Motorola, 1990.

[5] Norman C. Hutchinson, Shivakant Mishra, Larry L. Peterson, and Vicraj T. Thomas. Tools for implementing network protocols. *Software - Practice and Experience*, 19(9):895–916, 1989.

[6] Norman C. Hutchinson and Larry L. Peterson. Design of the *x*-kernel. In *Proceedings of ACM SIGCOMM '88*, pages 65 – 75, 1988.

[7] Norman C. Hutchinson and Larry L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[8] C. Maeda and Brian N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of 14th ACM Symposium on Operating Systems Principles*, December 1993.

[9] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX Conference*, pages 259–269, January 1993.

[10] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[11] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of 11th ACM Symposium on Operating Systems Principles*, pages 39–51, November 1987.

[12] J. B. Postel. User datagram protocol. *Request For Comments 768*, USC Information Science Institute, Marina Del Ray, CA, August 1980.

[13] J. B. Postel. Internet protocol. *Request For Comments 791*, USC Information Science Institute, Marina Del Ray, CA, September 1981.

[14] J. B. Postel. Internet control message protocol. *Request For Comments 792*, USC Information Science Institute, Marina Del Ray, CA, September 1981.

[15] J. B. Postel. Transmission control protocol. *Request For Comments 793*, USC Information Science Institute, Marina Del Ray, CA, September 1981.

[16] Thomas Pittman. Two-level hybrid interpreter/native code execution for combined space-time program efficiency. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 150–152, St. Paul, MN, June 1987. ACM.

[17] David. C. Plummer. An ethernet address resolution protocol. *Request For Comments 826*, DCP@MIT-MC, November 1982.

[18] D. Stuart Ritchie. The Raven kernel: A microkernel for shared memory multiprocessors. Technical Report 93-36, Department of Computer Science, University of British Columbia, April 1993.

[19] D. Stuart Ritchie and Gerald W. Neufeld. User level IPC and device management in the raven kernel. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, September 1993.

[20] David L. Tennenhouse. Layered multiplexing considered harmful. In *Proceedings of IFIP Workshop on Protocols for High-Speed Networks.* H. Rudin editor, North Holland Publishers, May 1989.

[21] C. A. Thekkath, T.D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, 1993.

[22] M. Yuhara, Brian N. Bershad, C. Maeda, and J. Eliot B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proceedings of the 1994 Winter USENIX Conference*, pages 153–165, January 1994.

# Device Driver Issues in High-Performance Networking

John Michael Tracey Arindam Banerji
*Distributed Computing Research Laboratory*
*Department of Computer Science and Engineering*
*University of Notre Dame*
*Notre Dame, IN 46556-5637*
*(219) 631-5273 (voice) (219) 631-9260 (FAX)*
*{jmt,axb}@cse.nd.edu*

## Abstract

High-performance networking requires attention to operating system support at the device driver level. Existing driver models, such as those of Unix, are not necessarily well-suited to supporting high-speed network interfaces. In fact, current drivers may represent a significant obstacle between applications and the high-speed network adapters they seek to exploit. Yet existing models cannot simply be discarded. Certain trends in RISC processor design can also tend to make existing device driver implementations less efficient. Specifically, many drivers make extensive use of operations which are becoming relatively more costly as RISC architectures evolve. This paper describes an effort currently underway to develop device drivers specifically designed to support high-speed network interfaces on RISC architectures. We have analyzed the operation of some existing commercial device driver implementations and modified one using several techniques. Taken together, these techniques promise to produce network device drivers which deliver the high level of performance demanded by today's high speed networks.

## 1. Introduction

The increasingly prevalent demands for high bandwidth, and in some cases low-latency network connectivity by a wealth of applications is well known. These growing application demands have been met by significant advancements in network hardware. Modern network architectures generally provide bandwidths at least an order of magnitude greater than traditional networks such as Token-Ring and Ethernet. Interconnections capable of providing gigabit bandwidths to the desktop over long distances have been demonstrated [Clark et al. 93]. A subset of the emerging architectures, including ATM, provide significantly improved latency characteristics as well [Cooper et al. 91].

Hardware, however, is only part of the picture. Without adequate software support, improved hardware capabilities remain underutilized or even inaccessible. Thus progress in hardware has been accompanied by advances in communication protocols and development of application programming interfaces (APIs) well suited to high bandwidth and low latency interconnections [Clark & Tennenhouse 90]. No less important than these components is the element which ties the software and hardware together, namely the device driver. But while protocols and APIs specifically designed for high-performance networks have been developed, the model after which device-drivers are patterned has remained largely static. The combination of these factors have increased the proclivity for network device drivers become the weak link in the performance chain [Banerji et al. 93].

This paper describes ongoing work to develop UNIX device drivers specifically designed to support high-performance network interfaces. The work began with a performance analysis of the Token-Ring, Ethernet, and Serial Optical Link device drivers of AIX 3.2. Subsequently, the operation of the Ethernet device driver was studied in much greater depth. Currently we are developing techniques to improve the device driver's performance. Ultimately, the techniques being developed to improve the Ethernet driver will be applied to the other drivers as well.

The remainder of this paper proceeds as follows. The next section describes the operation of the IBM Ethernet High-Performance LAN Adapter device driver of AIX 3.2.5 [IBM 93c]. This driver is taken as representative of current network device driver implementations in commercial UNIX operating systems. It is used as the initial basis for our modifications. Section 3. presents the proposed techniques for improving communication device driver performance. The emphasis is on techniques which are not specific to any particular communication protocol or network or machine architecture, but rather apply to computer networking in general. In section 4., we present the results we have achieved from the modifications we have made to date. Finally, in Section 5., we summarize our contributions.

## 2. Ethernet Device Driver Operation

The IBM Ethernet High-Performance LAN Adapter serves as an interface between the Micro Channel expansion bus and an Ethernet local area network. The adapter includes 16 k bytes of static RAM and a 16-bit microprocessor running at 12.5 MHz. It features a 32-bit bus master interface to the Micro Channel and has an on-board direct memory access (DMA) controller. The adapter is capable of either memory-mapped, or DMA modes of operation. The driver which supports the adapter in AIX 3.2 uses DMA.

Granted, application of the term "high-performance" to an Ethernet device no matter how advanced may be objectionable to those accustomed to modern networks such as FDDI and ATM. The operation of the High-Performance Ethernet adapter as pertains to its interaction with its host processor, however, is similar to that of many adapters used with higher bandwidth networks. Also, the operation and structure of the Ethernet driver epitomizes the classic Unix communication device driver model we seek to improve.

The source code for the device driver is segregated into two distinct components: a device-independent common I/O (CIO) portion, and an adapter specific part. A modified version of the CIO portion is also used in the device driver for IBM's integrated Ethernet adapter. The code appears to have been written as a general-purpose top half capable of being used for LAN adapter device drivers in general. The code is not reused, however, in either the Token-Ring High-Performance LAN adapter, or Serial Optical Link device drivers.[1] The CIO portion of the driver provides an interface between the high-level abstractions of the Unix device driver model and the low-level device-specific constructs. As with most Unix device drivers, the code can also be separated into device management and an interrupt handler portions, referred to as the top and bottom halves respectively.

The adapter utilizes two circularly-linked lists of buffer descriptors in its static RAM, one each for its transmit and receive lists. One such list is illustrated below in the upper portion of Figure 1. Each on-card buffer descriptor consists of four fields. The first is a pointer to the next descriptor in the circularly linked list. Next is a control/status field which allows the device driver and adapter to communicate relevant information concerning the described buffer's state. The third field contains a count of the number of bytes contained in the buffer. Finally, the buffer descriptor quite logically contains a pointer to the buffer itself. This pointer resides in the Micro Channel memory address space and gets translated to an address in the system memory address space during DMA operations. During its initialization, the device driver programs the system's

1. Unfortunately, we do not currently have the source code for either the FDDI or ATM drivers.

DMA controller to set up a static mapping between the buffer addresses in the Micro Channel memory address space used by the card and the addresses of the corresponding buffers in the system memory address space used by the device driver.

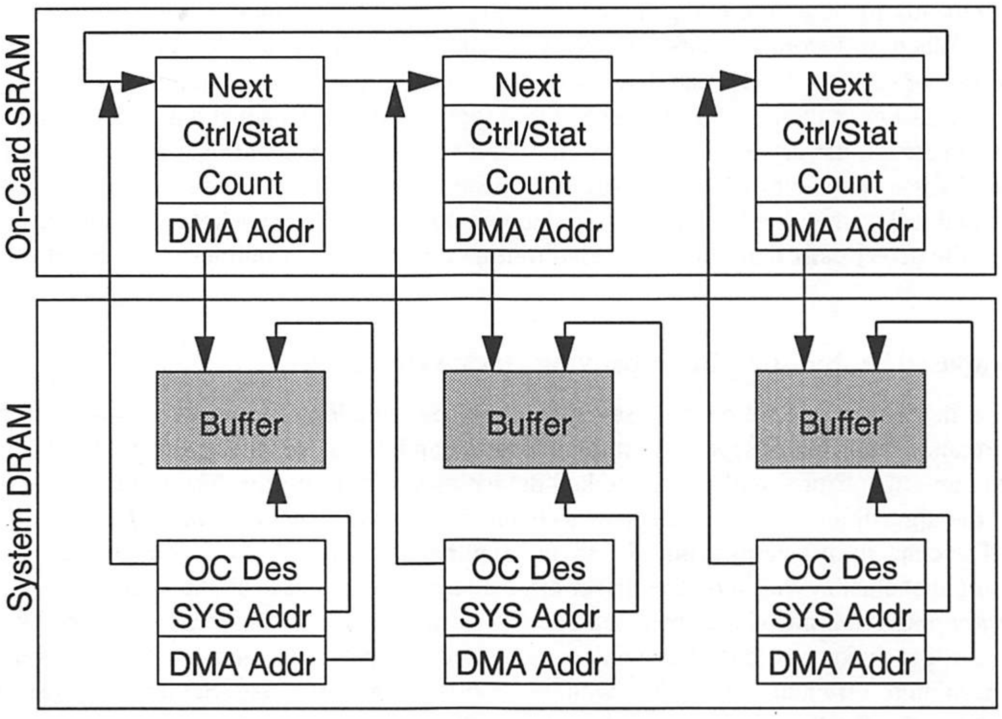


**Figure 1. - On-Card and System Buffer Descriptors**

Similarly, the device driver maintains two arrays of buffer descriptors in system memory, one each for transmission and reception. Each array of system buffer descriptors in system memory corresponds to a list of on-card buffer descriptors in the adapter's static RAM. The structure of these system buffer descriptors is illustrated in the lower portion of Figure 1. Each system buffer descriptor consists of three fields. The first is the offset from the beginning of the adapter card's static RAM of the corresponding on-card descriptor. A second field contains the address in the Micro Channel memory address space of the described buffer. This field contains the same value as the identical field in the corresponding descriptor on the card. The third and final field contains the address in the system memory address space of the described buffer.

To perform a transmit operation, the device driver copies the data to be transmitted into the next available transmit buffer in system memory and writes the appropriate value to the control/status field in the corresponding on-card descriptor. The card "sees" a new buffer to be sent and initiates a DMA operation to copy the data from the buffer in system memory to the adapter static RAM. Once transmission is complete or fails, the adapter informs the device driver via an interrupt. Upon reception of an incoming packet, the adapter copies the received data to the next available receive buffer in system memory via DMA, updates the control/status and count fields, and informs the driver via an interrupt.

Transmitted data is not copied directly to the dedicated low-level transmit buffers just described on its way from application code. Instead, the driver maintains a single high-level transmit queue. When a user-level client performs a write to the Ethernet device, the data to be transmitted is first copied from its initial location in the user address space to an mbuf [Leffler et al. 89] within the kernel address space. This mbuf is then appended to the single high-level transmit queue before being copied to a low-level transmit buffer. From there, the data is copied to the adapter static RAM via a DMA operation initiated by the adapter as described above. For a client writing to the Ethernet device from within the kernel, the original copy from user to kernel space is avoided as it is assumed the data already resides in an mbuf within the kernel address space. As with user-level clients, however, data written by clients within the kernel is appended to the high-level transmit queue before being copied to low-level transmit buffers.

When an incoming packet is received, the adapter immediately copies it from the static RAM on the card to the next available low-level receive buffer in system memory via DMA as described earlier. It then issues an interrupt to inform the driver a packet has arrived. Upon receiving the interrupt, the driver allocates an mbuf and copies the received data into it. If the packet is destined for a user-level client, the driver appends the mbuf to that client's high-level receive queue and wakes the client process if it is sleeping on a read. If the packet is for a client within the kernel, the driver calls that client's receive routine directly with a pointer to the mbuf as an argument.

## 3. Proposed Techniques for Improving Device Driver Performance

In this section, we propose several techniques intended to improve network device driver performance. The first concerns the implementation and use of watchdog timers which are manipulated along the critical paths of some key device driver entry points. The second technique concerns the algorithms used by the driver to manage its table of *network identifiers* (netids). [IBM 93c] The concept of a netid is introduced and explained in this section. Third we address the need for mutual exclusion within device driver critical sections and describe the technique of *optimistic interrupt protection* introduced by researchers at Carnegie Mellon University. [Stodolsky et al. 93] The fourth technique is that of providing optimized fast paths designed to handle given specific instances more efficiently than code which supports the general case. Saving the most important for last, we finally discuss the use of shared memory to eliminate the cost of copying data between the kernel and user address spaces.

## 3.1. Watchdog Timers

In order to detect adapter failures, the driver sets a watchdog timer every time it initiates an operation which the adapter must respond to. Such operations include packet transmission. Every time the driver passes a packet to the adapter to be transmitted, it starts a watchdog timer with a duration of ten seconds. Under normal operation, the driver stops the timer when it receives an interrupt indicating the transmission is complete. If the interrupt fails to arrive within ten seconds, the timer expires generating its own interrupt and alerting the driver of the error. Such use of watchdog timers is common in nearly all device drivers. Researchers have recognized the importance of reducing the overhead associated with timer management and developed efficient implementations. [Varghese & Lauck 87]

During periods of high transmit activity, the driver may set the watchdog timer to expire in ten seconds, thousands of times a second. This would seem unnecessary. It is, of course, not important that the timer expire in exactly ten seconds. Rather, this number has been chosen somewhat arbitrarily. It would be sufficient for the driver to be assured a timer would expire within some relatively broad range, say five to twenty seconds. It is also not essential that the driver stop the timer immediately when an anticipated interrupt arrives especially in cases where most of the timer's initial duration remains. (This case can be identified using a small number of instructions.) Instead it can simply set a flag indicating the timer is inactive and ignore its expiration in the unlikely event it expires before being stopped during the processing of a subsequent anticipated interrupt.

We thus propose the following technique for reducing the number of instances in which the driver starts and stops the watchdog timer. Instead of restarting the timer on every transmission, it checks to see if the timer is already set to expire within the acceptable range. It does so by reading the current time, subtracting the time at which the timer was started, and comparing the difference to some constant value. Similarly instead of stopping the timer each time an anticipated interrupt is received, the driver simply sets a flag indicating the timer is inactive, and checks to see if the timer has some minimum threshold duration remaining. If so, the driver refrains from stopping the timer. The cost of handling watchdog timer expirations which are ignored is expected to be offset by the reduced number of instances in which the timer is started and stopped.

## 3.2. Network ID Management

Before receiving frames from the Ethernet device, a client must perform a `start ioctl`. This call allows the client to specify a network identifier or netid to be associated with an opening of the device. The netid is a field in the Ethernet header, also known as the type field, which essentially identifies a network protocol. The term *service access point* or SAP is used for a field with the same purpose in other link level protocols. To give an example, IP packets are encapsulated in Ethernet frames with a netid field of 0x0800. Netware packets travelling over Ethernet have a netid value of 0x8137. When a frame arrives, the driver examines its netid field to determine which client, if any, should receive it.

Analysis of the unmodified Ethernet driver exposed one so-called optimization which may, in fact, decrease performance in some cases instead of enhancing it. When determining which client is to receive an inbound packet, the driver first checks to see if the packet is associated with the same netid as the last packet received. If so, the appropriate client is found immediately, eliminating the need to perform a linear search through a table of netids. This optimization is based on the idea that data for a given high-level protocol is often fragmented into multiple frames which results in a stream of frames for the same netid. Some netids, however, are used by protocols which generally, or always package messages within a single network packet. The netids associated with the ARP and RARP protocols on Ethernet are two examples. When the Ethernet driver receives an ARP packet, it assumes, the next frame received will contain an ARP request also which, in the common case is exactly the wrong assumption to make.

This optimization directly concerns the separation of protocol-specific knowledge from communication device drivers which is viewed by many [Peterson et al. 90] (including the authors) as a desirable property. Assuming the next packet will be destined for the same netid requires no protocol-specific knowledge. Assuming the packet immediately following an ARP request will be for a different netid rather than for the same, requires the device driver be contaminated with protocol-specific code. We are investigating the use of statistics such as the averaged number of packets received in a row for each particular netid as an improvement over the current scheme. This information along with more efficient search methods, such as a has function, may yield improvement.

We do not expect this modification to yield any significant performance improvement when subjected to practical use at least not in most environments where the vast majority of frames do, in fact, have the same netid as the frame which immediately proceeded them. In environments which do use multiple netids, however, the current scheme may tend to limit performance unnecessarily.

## 3.3. Optimistic Interrupt Protection

Researchers at Carnegie-Mellon University [Stodolsky et al. 93] have proposed an optimization which reduces overhead associated with assuring mutually exclusive access to critical sections of code by changing the processor interrupt level. Such interrupt level manipulation is a common technique for guaranteeing mutual exclusion. The problem with the technique, as pointed out by the researchers, is that the process of changing the interrupt level on a RISC processor is getting relatively more expensive as RISC implementations evolve. Increasing the desire to avoid this cost is the realization that it yields value only in the relatively uncommon, but admittedly important, case where a mutual exclusion would otherwise be violated due to an interrupt A method of ensuring mutual exclusion in the more common case where critical sections are not interrupted is desirable.

The idea proposed by the researchers is to set a software flag instead of changing the interrupt level and code interrupt service routines to check the software flag. If an interrupt service routine discovers it has interrupted an operation which should not be interrupted, it sets another flag which identifies the particular interrupt, suspends its own execution and resumes at the routine which was running when the interrupt occurred now with the interrupt priority raised to prevent nested interrupts. When the so-called uninterruptible sequence completes, it checks to see if it must continue a suspended interrupt. This optimization eliminates the need to pay the cost associated with changing the interrupt level for every sequence, when they are in fact rarely interrupted.

## 3.4. Device Driver Fastpaths

This technique is more of an overall approach, than a specific optimization. The approach is to provide *fast paths,* device driver entry points designed to efficiently handle specific situations. Fast paths allow a device driver client to use knowledge of its behavior to allow operations to be performed more efficiently than is possible in the general case. One example of a fast path in the existing Ethernet driver [IBM 93a] is the fast write routine. The driver supports an IOCTL operation which returns a pointer to a transmit routine. This IOCTL is only supported for clients within the kernel. The transmit routine, also known as the fast write routine, is optimized for kernel clients. It contains none of the code needed only to support user-level clients such as calls which copy data from user to kernel-level space. The existing fast write fast path supports the "special case"[1] of the device driver client is within the kernel.

Further performance improvements can be gained by supporting additional special cases with their own fast paths. One such fast path we are adding is intended to support transmission of very small messages with minimal latency. The need for this path was identified while developing a distributed shared memory facility for AIX. The performance of the facility relies on the ability to send small messages containing page requests quickly. The special purpose nature of the fast path allows a number of important optimizations to be made. Due to the priority of the message to be sent, and its small size, the routine bypasses the high-level transmit queue and copies the data to be sent directly to a dedicated low-level transmit buffer from which the adapter transfers data via DMA. In effect, the ability to bypass the high-level transmit queue implements a simplistic two-level priority scheme at the frame level. The routine either waits for a transmit buffer to become available, or depending on how the driver has been configured, is assured at least one transmit buffer is always available exclusively for its use. One final optimization is that the fast path routine assumes the packet to be sent is of a specific pre-determined length which eliminates the need for the length to be calculated within the transmit routine.

Implementation of fast path routines on the receive side are complicated by the inability to know what process or netid an incoming packet is destined for before it arrives. This inability does not, however, preclude the possibility of fast path, or at least faster path, receive routines. One consideration is that although it is impossible to tell *with certainty* what device driver client the next inbound packet will be destined for, it is often possible to use the expectation that a given client will be receiving a packet soon as when awaiting a response to a request. The benefits of assuming

1. In fact, the case of the device driver client being within the kernel is, by far, the most common.

the next packet received will be for a particular client must be weighed against the cost if the assumption is incorrect. We have not investigated the potential of this optimization.

It is a common case that the vast majority of inbound packets will be destined for a single device driver client, for example the network interface layer of IP on most Unix workstations. Even in this scenario it is possible to improve the performance of receive operations by eliminating code to support the general case when it is known to be unneeded. As an example, the receive routine of the Ethernet driver in AIX includes code to support *promiscuous mode* clients which receive a copy of every packet on the network. Despite the fact this mode is rarely used, the receive routine contains a check to see if there are any promiscuous mode clients for which a copy of the received packet must be made. Similar checks are made for packets sent to multicast addresses which represent another rarely-used feature. The simple strategy is to preclude the presence of such infrequently used code on the critical path when it is known to be used. The code to handle promiscuous clients, for example, can be removed from a commonly used receive routine and relocated to a specialized receive routine which is used only when the adapter has been put into promiscuous mode.

## 3.5. Shared Memory

The traditional device driver model requires user-level clients to incur the undesirable penalty associated with copying data between the user and kernel address spaces. Ideally, user-level clients should be afforded the same luxury in terms of avoiding that copy as are clients within the kernel. Two pertinent considerations highlight the need to eliminate this copy. The first is the emergence of multimedia applications happy to consume network bandwidth which is a significant fraction of the bandwidth available on the memory bus. This leads directly to the somewhat alarming conclusion that user-level clients should be able to create data directly in the kennel address space. The second is the widespread adoption of alternate operating system structures such as microkernels [Acetta et al. 86]. Such structuring often entails the prohibitively expensive practice of copying communication data across multiple (that is, more than two) protection domains. Given these considerations, we have adopted a shared I/O buffer management scheme[1] that extends from user-level applications to device-drivers.

The *shared heap buffer management* utility allows execution threads in different protection domains, uniform access to shared communication data. It is realized as a kernel-level pageable heap that gets mapped at the same addresses in all application-domain processes. This mapping is chosen to be the same as that used by kernel-domain entities. Thus, a uniform mapping is maintained across both protection domains. There are two major issues in the implementation of the heap - security and buffer allocation.

Simplistic security schemes, such as private areas within the shared heap, are not effective against malicious entities [Druschel & Peterson 93]. A page re-mapping scheme that mimics a MOVE operation is required. This entails dynamically changing the protection bits on the pages involved in a cross-domain transfer. Such an operation guarantees that malicious entities cannot corrupt shared communication data after it has been passed into the kernel. The only fool-proof solution is a page remapping mechanism based on per-process page tables. Unfortunately, most such implementations require some changes to the virtual memory management component of existing operating systems. However, in order to efficiently handle application such as multimedia, similar changes will become necessary in other operating system components as well [Rangan & Vin 91].

As is obvious, such page-remapping schemes introduce some overhead. Given that most messages on the internet are smaller than 200 bytes [Kay & Pasquale 93a] (memory-memory

1. This scheme is closely based upon the fbuf facility [Druschel & Peterson 93]

copying time for a 200 byte packet is approximately 1 μsec on a 25 Mhz RS/6000), it is almost certain that a page-remap scheme takes longer than a memory-memory copy for very small messages. However, this disadvantage can be offset by the allocation mechanisms provided by the shared heap utility.

Cross-domain allocation on the shared heap is separate but co-ordinated. The separation is required to ensure that allocation itself does not require a protection domain switch, such as a system call. The co-ordination guarantees that multiple allocators can work concurrently. This is implemented by two mechanisms - per-domain allocators that each manage a part of the heap and locking mechanisms based upon fast compare and swap[1] provided by AIX [IBM 93b]. Such mechanisms speed up allocation; however, allocation costs are fixed and not dependent upon the amount of data allocated. Hence, sophisticated allocation mechanisms that can reduce some of the data-touching [Kay & Pasquale 93b] overhead of transport protocols are necessary.

Specialized shared heap allocators are being constructed to utilize application and device specific information, in order to optimize memory management. One such allocator based upon the x-kernel [Peterson et al. 90] message objects, internally maintains non-contiguous chunks of data while allowing clients to treat the data as contiguous. Such non-contiguous chunks can then be easily mapped on to specific MTU sizes of a device, thus avoiding fragmentation.

## 4. Experience With Techniques for Improving Device Driver Performance

In this section, we report the results from the modifications whose performance we have analyzed to date. Several of the modifications remain to be fully tested. We start by describing the testing procedure. We then report our experience with watchdog timers, optimistic interrupt protection, address space reservation, and shared memory.

### 4.1. Testing Procedure

We used a single metric to evaluate device driver performance: round-trip latency between user-level clients. In order to measure this value, we ran a pair of user-level processes, an *initiator* and an *echo* process, on two machines attached to the same Ethernet segment. Each process opened the Ethernet device ("`/dev/ent0`") using the UNIX `open` system call and set the network identifier it would receive packets for by issuing an `ioctl` call. The initiator process then recorded the current time and sent a packet of sixty bytes in length, including the Ethernet header, to the echo process on the remote machine. Sixty bytes was chosen because it is the minimum size allowed for an Ethernet packet. Upon receiving the packet, the echo process would respond by sending a packet of the same length back to the initiator which would record the current time immediately after receiving the reply.

Before starting the test, both processes wrote the source Ethernet addresses and destination network identifier fields of the Ethernet header at the beginning of the message buffer. The initiator process filled in the destination Ethernet address of the first packet is sent to the echo process. Subsequently, each process copied the source Ethernet address from the last packet it had received into the destination Ethernet address of the next packet to be sent. Data transmission was initiated using the `write` system call.

In order to reduce the effects of caches, virtual memory translation, and interference from other device driver clients, the round trip latency test was performed twice in immediate succession. Upon receiving the first packet back from the echo process, the initiator would immediately send a second which would also get echoed back. The round-trip latency values were computed and recorded for both round trips. The value from the second round-trip was used in our perfor-

1. Such mechanisms are available on many systems.

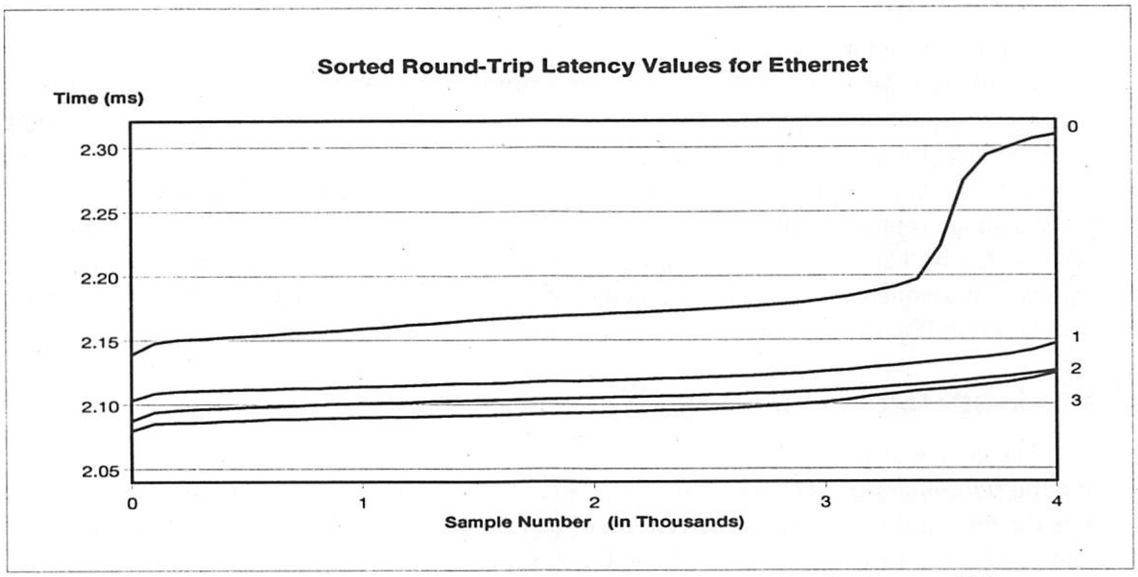


**Figure 2. - Effect of Various Driver Modifications on Round-Trip Latency Values**

mance analysis. We intended the first packet to "warm up the path" to be traveled by the second. Apparently it did since we found the round-trip latency value for the second packet to be approximately five to seven percent less than the one sent immediately before it on average. This difference persisted and remained relatively constant as we applied various modifications to the driver. The five to seven percent difference is particularly significant when compared to the impact of the individual modifications we were testing some of which were in the range of only one percent.

For each version of the driver which we tested, we repeated the double round-trip trial 5 k ($5*2^{10}$) times with a delay of 0.5 seconds between trials. Each test was therefore performed over a period of approximately 43 minutes. We observed that although most of the 5 k values fell within a narrow range, some small subset of them were an order of magnitude larger. We believe these drastically larger values to be due to such extraneous factors such as collisions with other network traffic and arrivals of unrelated device interrupts at inopportune times. In order to focus on the contribution to the round-trip latency made by the driver, we sorted all 5 k values and took the arithmetic mean of the lowest 4000 to produce our representative round-trip latency figure.

The tests were performed on a pair of IBM RS/6000 Model 530s (running at 25 MHz) equipped with IBM Ethernet High-Performance LAN adapters. Both machines were running AIX 3.2.5 in multi-user mode with a somewhat reduced complement of background processes running including AFS, NFS, and NIS. The initiator and echo processes were the only active user-level processes on the machines. They were run by root using the nice command (`nice -20`) to give them the highest possible user-level priority. The vital product data for the Ethernet adapters on the initiator and echo machines respectively was: part numbers 071F1182 and 00063369, engineering change numbers C73857 and C73859, read only storage levels 0028 and 0015, device driver levels 00 and 01, field replaceable unit numbers 081F7913 and 00063368, serial numbers 90002403 and 00116753, manufacturers 204491 and 204491.

Figure 2. shows a graph of the lowest 4000 sorted round-trip latency values for various versions of the driver. The top line, labeled 0, represents the original unmodified driver from AIX 3.2.5. The lowest latency value obtained with the original driver was 2.14 ms. The average of the lowest 4000 values was 2.18 ms. As can be seen on the graph, the vast majority of the latency values fall between 2.15 ms to 2.20 ms. The highest 500 or so values, however, display a sharp increase with some values in excess of 2.30 ms. The remaining lines on the graph are explained below.

## 4.2. Watchdog Timers

In trying to improve the performance of the driver by reducing overhead associated with timer manipulation, we assumed the overhead was sufficient to warrant reduction. That is, we proceeded on the assumption the small cost of determining whether to start or stop the timer would be less than simply doing so. That assumption proved false. When we modified the driver to determine whether or not it should manipulate the timer, its performance got slightly worse. Specifically, the average round-trip latency increased 22.3 μsec and 18.9 μsec for the average and shortest cases respectively. This represents an increase in latency of approximately one percent. The modification was subsequently dropped. The results of performance tests made with the modification are not shown in Figure 2.

## 4.3. Optimistic Interrupt Protection

The device driver must ensure mutually exclusive access to data structures shared between its *top* and *bottom halves*. The top half of the driver consists of entry points called by processes such as the read and write routines. The bottom half consists of service routines which are run in response to device interrupts. These include the routines which run when a packet is received or a transmission completes. The original driver ensures mutual exclusion by using the AIX `i_disable` and `i_enable` system calls to temporarily raise the processor interrupt level. These calls provide the same functionality as the `spl` call found in many other UNIX implementations. For example, when a user-level process performs a read, the driver raises the interrupt level before accessing that process' receive queue in order to prevent an incoming frame from being concurrently appended to it by a routine handling a receive interrupt.

We sought to improve driver performance by eliminating use of the `i_disable` and `i_enable` calls from the read and write paths. In order to assure mutual exclusion, we use a routine hand coded in assembly language which performs an atomic test and set operation.[1] We call this routine to set a flag which protects the set of shared data structures being manipulated. In the simple and frequent case where the flag can be set the code simply proceeds as though it had changed the interrupt level. The case where a flag can not be obtained, indicating another thread is already active within the critical section requires more consideration.

The case of the write path was relatively simple to handle. We added a mutual exclusion flag, `xmit_mutex`, which is used to relegate access to transmit data structures such as the high-level transmit queue and low-level transmit buffer descriptors. There are three routines which need to set the flag. One is the user-level write routine. If this routine is unable to obtain the flag, it simply spins trying to do so until it succeeds. This method would fail miserably in kernel implementations where system calls are not interruptible. In such cases the routine could be coded to sleep on an event instead. The second routine is the fast write routine exported only to kernel-level clients. If this routine fails to obtain the flag on its first try, it immediately returns with an error code indicating the operation should be retried. This is exactly what the routine does if it encounters other temporary error conditions such as the transmit queue being full. The third routine handles interrupts generated when a transmission completes. This routine checks to see if there are elements on the high-level transmit queue which were waiting for transmit-buffers which are available now that a transmission is complete. If this routine is unable to claim the flag it simply does not process the elements waiting on the transmit queue. This is permissible since whatever process has already claimed the flag either has or will.

The case of the receive path was somewhat more involved. We associated a mutual exclusion flag with each user-level client's receive queue. There are only two routines which to set this

1. The POWER architecture of the RS/6000 does not include any instructions which perform an atomic test and set operation. Interestingly, the POWERPC architecture does.

flag, the user-level receive routine, and the receive interrupt handler. As with the user-level write routine, if the user-level receive routine fails to obtain the flag on its first try, it simply spins attempting to do so until it eventually succeeds. If the receive interrupt handler fails to obtain the flag, however, it cannot simply spin as it would do so forever. Instead, the routine sets a flag indicating it was unable to claim the flag, saves a pointer to the received data, calls the AIX `i_mask` kernel service to mask off the interrupt associated with reception, and returns. This allows whatever user-level process possesses the flag to continue.Before that process releases the flag, it checks to see if the interrupt handler was denied access to the receive queue. If so, that process resets the flag indicating the interrupt handler was denied, processes the received data, can reenables receive interrupts by calling the AIX `i_unmask` kernel service.

Elimination of the `i_disable` and `i_enable` calls yielded encouraging results. The second line from the top, labeled 1, in Figure 2. shows the test results from a version of the driver modified to eliminate the interrupt priority manipulation calls from the write path. The lowest and average latency values obtained with this driver were 2.10 ms and 2.12 ms respectively. These represent an improvement of 36 μsec and 62 μsec or 1.7% and 2.9% in the lowest and average cases. The third line from the top, labeled 2, in Figure 2. represents the test results from a driver which was also modified to eliminate the interrupt priority manipulation calls from the read path. This driver displayed a further improvement of 16 μsec and 14 μsec in the shortest and average cases respectively bringing the total improvement to 2.4% and 3.5%.

We believe the greater improvement in eliminating the interrupt priority manipulation calls from the write path is due to those calls being on the critical path for a write, but occurring before data had arrived in the case of a read. Besides producing lower latency values, the elimination of the calls appears to have eliminated the spike in latency values encountered with the original driver. We suspect this may be due to a reduction in the length of the overall code path, but have not confirmed this.

### 4.4. Address Space Reservation

Before sending or receiving a frame, the driver must first obtain access to the adapter's memory and I/O ports. It does so by invoking the AIX `vm_att` kernel service which attaches the bus memory and I/O address regions to the process' or interrupt handler's address space. In order to evaluate the overhead associated with this operation, we modified the AIX kernel to reserve a single segment register which would always provide access to the necessary regions. The fourth line from the top in Figure 2., labeled 3, represents the test results of a driver modified to eliminate the calls to `vm_att` from the send and receive paths. This modification yielded a very slight improvement of 8 μsec and 9 μsec in the shortest and average cases. These values represent less than 0.5% of the round-trip times for the original driver.

### 4.5. Shared Memory

Comprehensive test results for the shared memory modifications were not availible at the time of publication. See section 6. for the address from which the latest version of the paper can be obtained.

## 5. Conclusion

Although still at a preliminary stage, this work amply points out two significant facts. Although, considerable effort has been directed at optimizing protocol realizations, device drivers need to be implemented to adequately deal with high-speed network interfaces. Furthermore, one single optimization cannot ameliorate existing device driver woes, but a synthesis of distinct implementation techniques are necessary.

We are continuing to implement the proposed optimizations, most notably, the addition of support for shared-memory. We intend to apply the techniques described here to other network device drivers and wish to repeat the tests reported here on faster machines.

## 6. Availability

The latest version of this paper is available in Postscript form via anonymous ftp at invaders.dcrl.nd.edu under the directory /pub/TechReports.

## 7. References

[Acetta et al. 86]Acetta, M. et. al. "MACH: A New kernel Foundation for Kernel Development." In *Proceedings of the Summer 1986 USENIX Conference* (Atlanta, GA. July). The USENIX Association, Berkeley, CA, 1986, pp. 93-112.

[Banerji et al. 93]Banerji A. et al, "High-Performance Distributed Shared Memory Substrate for Workstation Clusters" In *Proceedings of the Second International Symposium on High Performance Distributed Computing* (Spokane, WA. July 20-23). IEEE Computer Society Press, Los Alamitos, CA, 1993. also Technical Report 93-1, Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, January 1993.

[Clark & Tennenhouse 90] Clark, D. D. and Tennenhouse, D. L. "Architectural Considerations for a New Generation of Protocols." In *Proceedings of ACM SIGCOMM*, (Philadelphia, PN. Sept.). 1990.

[Clark et al. 93] Clark, D. D. et al. "The AURORA Gigabit Testbed." *Computer Networks and ISDN Systems*, Vol. 25, No. 6, January 1993, pp. 599-621.

[Cooper et al. 91]Cooper, E. et al. "Host Interface Design for ATM LANs." In *Proceedings of the 16th Conference on Local Computer Networks* (Minneapolis, MN. Oct. 14-17). 1991, pp. 247-258.

[Druschel & Peterson 93]Druschel, P. and Peterson, L. L. "Fbufs - A High Performance Cross-Domain Data Transfer Facility." In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles* (Asheville, NC. Dec. 5-8). ACM Press, New York, NT, 1993, pp. 189-202.

[IBM 93a] *AIX Version 3 for RISC System/6000 - Kernel Extensions and Device Support Programming Concepts* (Seventh Edition), IBM Corporation, October 1993. (Part no. SC23-2207-00)

[IBM 93b] *AIX Version 3 for RISC System/6000 -Commands Reference Volume 1* (Seventh Edition), IBM Corporation, October 1993. (Part no. SC23-2233)

[IBM 93c] AIX Version 3.2 for RISC System/6000 - Technical Reference: Kernel and Subsystems Volume 5 (Seventh Edition), IBM Corporation, October 1993. (Part no. SC23-2386)

[Kay & Pasquale 93a] Kay, J. and Pasquale, J. "Measurement, Analysis and Improvement of UDP/IP Throughput for the DECstation 5000." In Proceedings of the Winter 1993 USENIX Conference, pp 249-258, January 1993.

[Kay & Pasquale 93b] Kay, J. and Pasquale, J. "The Importance of Non-Data Touching Processing Overheads in TCP/IP." ACM SIGCOMM, September 1993.

[Leffler et al. 89]Leffler, J. et al. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. New York: Addison-Wesley, 1989.

[Peterson et al. 90] Peterson, L. et. al., "The x-kernel: A Platform for Accessing Internet Resources." *IEEE Computer*, 23 (5), May 1990, pp. 23-33.

[Rangan & Vin 91] Rangan, P. V. and Vin, H. M. "Designing File Systems for Digital Video and Audio" In Proceedings of the Thirteenth ACM Symposium on Operating System Principles (Pacific Grove, CA. Oct. 13-16). ACM Press, New York, NY, 1991, pp. 81-94.

[Stodolsky et al. 93] Stodolsky et. al. "Fast Interrupt Priority Management in Operating System Kernels." In *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures* (San Diego, CA. Sept. 20, 21). The USENIX Association, Berkeley, CA, 1993, pp. 105-110.

[Varghese & Lauck 87] Vargheese. G. and Lauck, T. "Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of Timer Facility." In *Proceedings of the Eleventh ACM Symposium on Operating System Principles* (Austin, TX. Nov. 8-11). ACM Press, New York, NY, 1987, pp. 25-38.

# TCP/IP and HIPPI Performance in the CASA Gigabit Testbed *

Bilal Chinoy
*San Diego Supercomputer Center*
*bac@sdsc.edu*

Kevin Fall
*San Diego Supercomputer Center* †
*kfall@cs.ucsd.edu*

**Abstract**

We investigate the packet delay and loss characteristics of the wide-area HIPPI-based CASA gigabit testbed. Developed for high–speed local area device interconnects, HIPPI is a point–to–point, connection–oriented protocol. We show HIPPI blocking can degrade performance by increasing delay and/or packet loss. In the CASA network under conditions of blocking, a tradeoff exists between packet loss and delay variance. The tradeoff point is determined by a combination of factors: source packet rate, mean blocking rate, and a *configurable* connection establishment timeout threshold. We show the delay/loss tradeoff manifests itself in TCP as inducing either the *slow-start* algorithm or requiring TCP to adjust retransmission timeout values due to increased delay variance. In the former case, we have measured TCP throughput to drop by as much as 97% of its unblocked maximum, as compared with 75% for the latter case.

## 1 Introduction

The desire to interconnect high–speed peripherals and supercomputers has lead to the development of several link–layer protocols offering bandwidths above 100MBytes/s. Among these, the High Performance Parallel Interface (HIPPI) [6], as specified by the ANSI X3T9 Standards Committee, has enjoyed great popularity due to the availability of HIPPI interfaces for most supercomputers and high–speed peripheral equipment. HIPPI enables computers and peripheral devices to communicate at bandwidths of 800(1600) Mb/s over a 64(128) wire parallel cable, with a maximum distance of 25 m.

In an effort to provide a prototype distributed supercomputer computation environment, the CASA gigabit testbed was formed [5]. At present, CASA links geographically distributed HIPPI networks some 100 miles distant using gateways capable of encapsulating HIPPI data over fiber-optic SONET[12] links at 800Mb/s. In this paper, we describe the delay and loss characteristics of HIPPI networks as extended to a wide area in the CASA network. In section 2 we describe the motivation for examining the HIPPI protocol dynamics in the CASA environment. We then give a brief discussion of the operation of a conventional HIPPI network in section 3. In section 4 we describe the CASA network and how it differs from a conventional HIPPI local–area network. We then develop a model of port contention or *blocking* in section 3.2, and evaluate the effects of blocking in sections 6 and 7. Section 8 concludes.

*Supported by National Science Foundation and the Advanced Research Projects Agency under Cooperative Agreement NCR-8919038 with the Corporation for National Research Initiatives.

†Kevin Fall is also a member of the Computer Systems Laboratory, University of California, San Diego

## 2 Motivation

An important goal for the CASA effort is to understand key factors contributing to the performance of applications distributed across multiple supercomputers. Applications are divided into several *subproblems*, and executed on architectures most appropriate to the computational needs of each subproblem[1]

Supercomputer applications divided into subproblems may overlap computation with network communication [8]. The effect of network latency can be completely hidden if subproblem execution times are carefully matched to network speed. Achieving this *latency hiding* is critical to the performance of distributed applications.

*Blocking* may occur because HIPPI is a connection–oriented point-to-point protocol allowing no simultaneous connections. Blocking implies data may not be delivered to a destination because of a competing (unrelated) concurrent connection. By affecting delay and packet loss, synchronization between application subproblems may be lost, eventually affecting overall application performance as measured in total wall-clock time. Our goal in this study is to understand and quantify the effect of HIPPI blocking on the delay and loss of TCP/IP traffic.

## 3 HIPPI Overview

A HIPPI local area network typically consists of a crossbar switch with nodes attached in a star configuration. Each switch attachment point provides a distinct *input* and *output* port (64-wire cable). Connections may be initiated by an attached HIPPI node through a switch's input port to a specified output port. Connections are simplex in all cases, and may be set up and torn down rapidly. The switch itself is a passive device; it does not introduce delay after a connection has been established between two nodes.

### 3.1 HIPPI Protocol Description

The HIPPI protocol is *simplex* and *connection oriented.* Communicating HIPPI nodes must explicitly set up, manage, and tear down a connection. Bidirectional communication requires each node to act as both a source and destination. A HIPPI source may request a connection to a destination by raising the *REQUEST* electrical signal while simultaneously placing *connection control information* (*CCI*), including the address of the destination, on the data lines. The cross–point switch decodes the *CCI*, determines the correct output port, and attempts to connect the source and destination ports. If the destination port is in use by another active connection or the destination node refuses connections, the source receives a connection rejected message and no connection is established.

If a connection is established, the destination node acknowledges and accepts the connection by raising the *CONNECT* electrical signal. Either side may terminate the current connection at any time by deasserting either the *REQUEST* signals (at the source) or *CONNECT* (at the destination). Connections otherwise remain open indefinitely. Generally, a connection set-up requires a round trip time (RTT) delay, amounting to only about half a microsecond[2] when cable distances are limited to 25 m. Once a connection has been established, the switch is not directly involved in data delivery.

HIPPI delivers data in arbitrarily long *frames.* Frames are divided into a sequence of "full–sized" *bursts* and possibly a single "short" burst. Full–sized bursts contain 256 32-bit words of data each. Once a connection has been established, bursts of data are sent from the source port to the destination port, subject to flow control. The *READY* signal provides flow–control signaling from a HIPPI destination to its peer source. No round trip cable delays are incurred in the sending of bursts. A destination node may generate up to 63 *look–ahead* READY signals avoiding throughput degradation experienced by stop–and–wait operation.

[1] For example, a parallel scalar computation such as a lattice-gauge problem may be best implemented on a massively parallel machine and a floating point matrix multiply may be best executed on a fast vector machine.

[2] This value represents round trip delay and hardware overhead only.

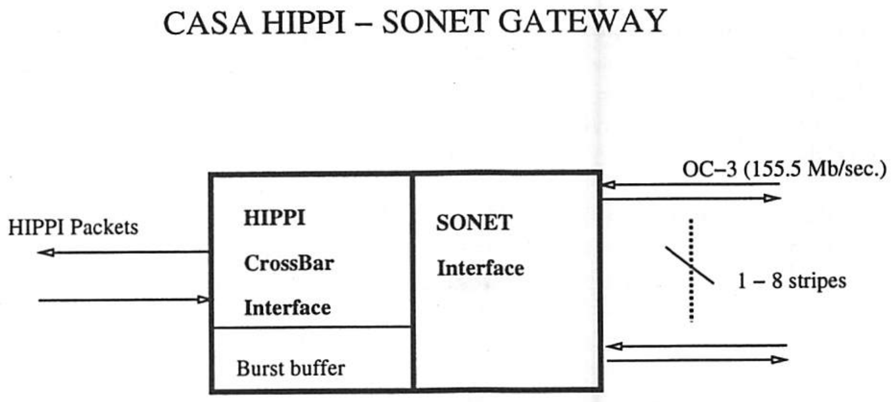


Figure 1: HIPPI–SONET Gateway

The *HIPPI-SONET* gateway extends HIPPI LANs across wide areas using SONET OC-3 links. It provides *local termination* of HIPPI connections, avoiding performance degradation caused by long round–trip delays. It includes 4MB of *burst buffers* to accommodate wide–area bandwidth-delay products.

## 3.2 Blocking Behavior

When a HIPPI destination has an active connection, no other connections to it can be established. An arriving request destined for a busy destination is rejected. Failure to establish a connection to a busy destination is called *blocking*; the requesting source is said to be *blocked*. The HIPPI protocol allows for arbitrarily long data packets, implying connections may be active indefinitely. In practice, packets are of finite length due to application or network layer framing. In the case of IP [2], the maximum size of the datagram is limited to 64 KB, which requires a 640 microsecond transmission time on a 800 Mb/s HIPPI network. However, an application using HIPPI directly (*raw* HIPPI) may send much larger packets, resulting in a correspondingly longer connection time. Applications displaying graphics output to a HIPPI–attached frame buffer are typical of the latter category.

A blocked source may retry to access a busy destination. The retry mechanism is configurable on a per–connection basis. With the *distributed* connection management policy, the source itself retries immediately or after some (usually random) time to establish a new connection. In the *centralized* policy, the switch itself arbitrates between competing connection requests to a single destination.

The HIPPI protocol specification supports the centralized policy by having sources set the *camp-on* (CO) bit in the connection request. Presence of the CO bit in a connection request signals the switch to cache the request and service it as soon as the requested destination becomes free. Camp-on is accomplished electrically by asserting the source *CONNECT* signal. The HIPPI–SC specification [6] does not specify a particular arbitration algorithm to be used when multiple sources are camped on a single busy destination.

# 4 The CASA Gigabit Testbed

The principal objective of the CASA gigabit testbed project is to build a distributed heterogeneous supercomputing environment supporting execution of large-scale scientific applications [5]. Because many supercomputers already have HIPPI interfaces and are part of local HIPPI networks, the CASA network was designed to interconnect geographically distant HIPPI networks using available fiber-optic technology.

A special purpose device called a *HIPPI-SONET* gateway (HSG) [9], has been designed and built by the Los

Alamos National Laboratory, for the CASA project, and is illustrated in Figure 1. An HSG is connected to a local HIPPI crossbar-based network and accepts packets destined for remote HIPPI nodes from local nodes. Incoming HIPPI data from the LAN is inserted into the *payload* portion of a SONET stream. A companion gateway at the remote end of the SONET link extracts the payload portion from the SONET stream, repackages the data as HIPPI frames and tries to deliver the frames to attached destination node.

The CASA wide-area link must be of at least 800 Mb/s bandwidth to ensure full rate HIPPI interconnectivity. Current SONET interface technology is not mature enough to provide a single fiber circuit at HIPPI speeds. Consequently, the CASA approach is to *stripe* HIPPI data across multiple circuits. The HSG gateways can stripe across up to 8 OC-3 links[3] each providing 155.5 Mb/s bandwidth.

## 4.1 Flow Control and Buffer Management

A destination HIPPI node asserts the *READY* signal, permitting the source to transmit data. The simple stop–and–wait nature of this flow control protocol is effective only because of small signal latencies. If READY signals are delayed while traversing long distances, the stop–and–wait protocol would be unable to fully utilize the available capacity. The HSG provides *local termination* of wide-area HIPPI connections, eliminating the need for end–to–end *READY* signaling while maintaining end-to-end transport of HIPPI frames.

A HIPPI conversation between remote machines across CASA is actually composed of 3 separate connections. In order to transmit data, the source device sets up a connection with the local HSG. Data flows across a *pseudo-HIPPI* connection between the pair of HSGs. No HIPPI signals are passed over a pseudo-HIPPI connection. Data arriving at the remote end is delivered by the destination HSG via a conventional HIPPI connection to the destination node.

The local HSG generates *READY* signals to the source node and sends data to the remote HSG across the SONET link. The remote HSG must have enough buffer capacity to queue received data while it attempts to establish a connection to the destination node.

Two interesting queueing scenarios arise in the long-distance CASA network:

- When data arrives at a remote HSG faster than it can be delivered to destination nodes, the remote HSG stores data in *burst buffers.* A local HSG flow-controls a source node whenever the remote HSG signals burst buffer saturation. Local HSGs may flow-control a source by either throttling the generation of *READY* signals or refusing new connections. Flow-control causes delays while packets wait in device driver and hardware queues.
- When a remote HSG attempts to establish a HIPPI connection to a destination and fails, it buffers any packets associated with the pending connection and retries. Packets arriving during this *retry* period experience *head-of-line* queueing. The *gateway hold parameter* $h$ defines the HSG retry period. All queued packets associated with the pending connection are discarded if the hold parameter timer expires. The hold parameter may be set to $\infty$, implying the HSG retries forever.

In the following sections we explore the consequences of these scenarios.

## 5 A HIPPI Blocking Model

HIPPI destination blocking occurs during periods of contention for a node's input port. We call an end–to–end conversation being degraded by HIPPI blocking the *blocked* connection. The sender on the blocked connection transmits packets at a deterministic rate $\lambda$. For our environment, there is a one–to–one correspondence between

[3]The OC designation in the SONET standard stands for Optical Carrier. The basic multiplexing building block for SONET is OC-1, which is 51.84 Mb/s.

packets and HIPPI connections because all nodes agree by convention to break connections between packets. We shall call a simplex connection competing with the blocked connection the *blocking* connection. The blocking connection keeps the blocked connection's receive port busy for a set of time intervals given by an exponential distribution with parameter $\mu$. For a mathematical model of HIPPI switches, the reader is referred to [4].

The packet (connection) interarrival rate is deterministic with parameter $\lambda$, and the service rate is equivalent to the blocking connection interarrival rate $\mu$. The CASA network will operate in one of two modes when $\lambda > \mu$, based on the configurable parameter $h$. For $h = \infty$, the HSG will only discard packets when burst buffer space is exhausted. When $0 < h < \infty$ (finite $h$), the HSG will attempt to deliver a HIPPI packet to the proper destination for at most $20h$ microseconds.[4] In summary, when $\lambda > \mu$ and $h = \infty$, buffer occupancy will grow (and thus introduce a larger mean packet delay) in proportion to $\lambda$ until packets are discarded. Alternatively, when $\lambda > \mu$ and $h$ is finite, buffer occupancy will remain bounded in proportion to $h$ but packet discards will increase in proportion to $\mu$.

We now turn to the expected effects of HIPPI blocking on TCP. We first review TCP's congestion control and retransmission timer setting policies. A complete description of the TCP congestion control scheme is introduced by Jacobson in [1]. A simulation study of the congestion control scheme with multiple TCP connections is presented in [10].

In TCP, packet loss is interpreted as an indication of network congestion. TCP assumes packet loss has occurred when either a retransmission timer has expired, or a number of duplicate acknowledgements have been received. In the first case, *slow–start* is initiated to reduce network load quickly, followed by a recovery period to re-establish a point of equilibrium in which packet injection rate matches packet removal rate. In the latter case, *congestion avoidance* reduces the sending rate, but not as drastically as slow-start. The TCP congestion control algorithms are realized by introducing a congestion window *cwnd* in the sender in addition to the normally–maintained window advertised by the receiver for flow control (*advertised_window*). The current send window is always set as MIN(*advertised_window*, *cwnd*).

*Slow–start* encompasses two techniques. It reduces *cwnd* to a single segment after a retransmission timer has expired. To recover, it includes an *additive increase* policy in which the send window is increased by one segment for each ACK received. Thus, slow–start grows the congestion window exponentially with respect to time, assuming no segments are lost. Slow–start operation is maintained at its exponential growth rate until the *cwnd* reaches the "slow start threshold" *ssthresh*. After *ssthresh* has been reached, the congestion window is grown linearly by incrementing it each time a window's worth of ACKs is received. In the event of retransmission timeout expiration, *ssthresh* is set to one half the previous window.

TCP enters *congestion avoidance* when a small number[5] of duplicate acknowledgements for the same segment have been received. Here, the current window is reduced by half, and *ssthresh* is reassigned to this new value (not to 1, which was the case for slow–start).

The above algorithms are actually independent and serve distinct functions. The slow–start algorithm attempts to "ramp–up" a sender's rate from zero until the available channel capacity is utilized. Slow–start is run only after a retransmission timer has expired (or a new connection is being initiated) when the channel is devoid of traffic. Congestion avoidance is run in all other cases, usually when the channel contains packets in transit. Its purpose is to track fluctuations in available channel bandwidth and modulate send rate slowly.

TCP sets segment retransmission timers according to an estimator taking both RTT mean and variation into account. When retransmissions do occur, subsequent timeout values are assigned exponentially. Estimating the RTT variance allows TCP to adapt its retransmission timer appropriately to a system with a large variation in RTT (especially useful in a multiaccess wide area network). The exponential backoff helps to avoid excessive retransmissions in lossy networks.

HIPPI blocking as described above can affect TCP by either blocking data segments (*segment blocking*) or ACKs

[4] The value $20h$ represents the amount of wallclock time required for a hardware countdown timer to reach zero if initialized with value $h$.

[5] This value, *tcprexmtthresh*, is typically set to 3.

(*ACK blocking*). We pursue the following intuitive description:

When segments are delayed but not discarded ($h = \infty$), both RTT mean and RTT variance increase. Acknowledgements are still delivered quickly, so the ACK return rate will closely reflect the trouble the data segments are experiencing, resulting in an accurate "ACK clock" . Spurious retransmissions should thus be limited. For a finite $h$, segments may be discarded, resulting in retransmission timers expiring (which will induce slow-start behavior and thus reduce throughput). When ACK blocking occurs, ACKs experience increased delay mean and variance when $h = \infty$ but high loss and bounded delay when $h$ is finite.

In effect, segment or ACK loss should affect a sending TCP similarly depending on the parameter $h$. We expect buffer overrun to be less frequent in ACK blocking due to the smaller packet size of ACKs during a unidirectional TCP data flow. These expectations are evaluated in the next sections.

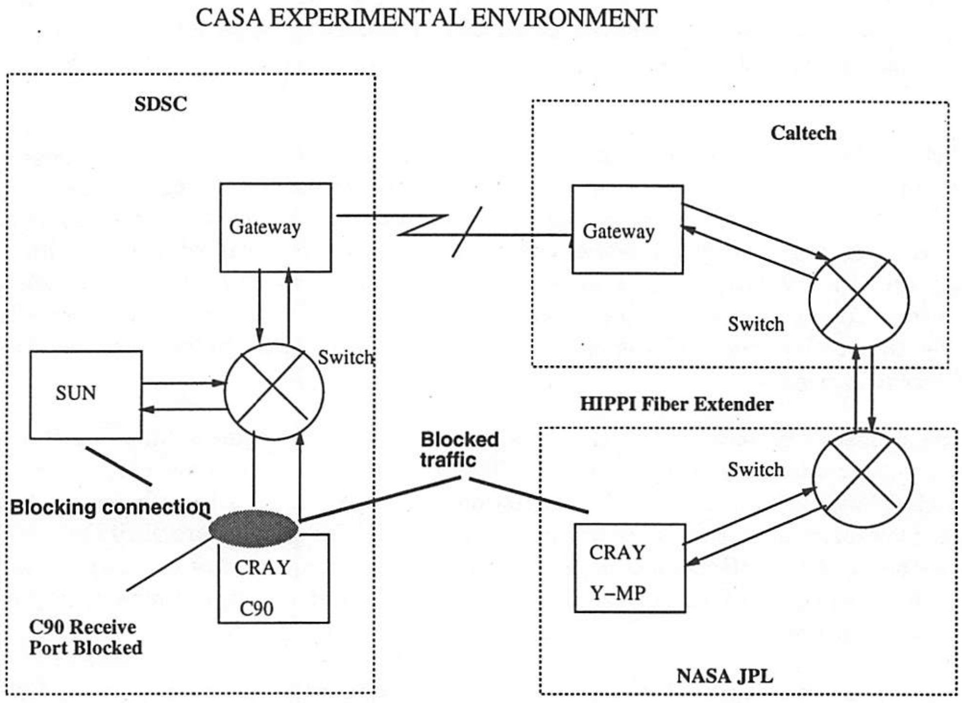


Figure 2: Experimental Configuration

A HIPPI wide-area connection (*blocked* connection) is established between the SDSC Cray C90 and JPL Cray YMP. A Sun workstation attached to the SDSC HIPPI LAN acts as a *blocking* connection source.

## 6 Experimental Environment

Our experimental configuration is shown in Figure 2. At the SDSC site, we have a Cray C90 and a Sun workstation on the HIPPI LAN based on the NSC PS–32 HIPPI crossbar switch [7]. We used 4 out of the 8 OC-3 links between SDSC and CalTech, providing an aggregate bandwidth of approximately 620 Mb/s.[6]

The CalTech site has two supercomputers connected to the HIPPI LAN, but neither of these are currently capable of using the TCP/IP protocol suite. CalTech is linked to JPL via a HIPPI fiber extender.[7] The extenders provide

[6] We were limited to 4 stripes when the experiments were performed.

[7] A HIPPI fiber extender is a device capable of transmitting all the HIPPI electrical signals across a fiber-optic link. The use of this

transparent access between CalTech and JPL.

We create a blocked connection between the JPL Y-MP and the SDSC C90. On the blocked connection, we vary both the packet size and the packet source rate. We choose two packet sizes: 64bytes and 64 Kbytes, representing typical cases of interactive and bulk data transfer, respectively. We choose three packet rates: 1 pps, 10 pps and 100 pps. Round trip packet delay and loss were measured using the *ping* program on the SDSC C90.

The Sun workstation is used to create the blocking connection by establishing HIPPI connections to the SDSC C90 for a random time period. The blocking connection is modeled by selecting the blocking time as an exponentially distributed random variable with parameter $\mu$. We term the mean blocking time *blocking factor* ($BF = 1/\mu$), and vary it between 0.001 sec. and 1 sec. We choose two values for the gateway hold time $h$: $\infty$ and 10 ms. For the TCP blocking study, we establish a bulk-data transfer with ACK blocking between the JPL Y-MP and the SDSC C90. The *File Transfer Protocol* (FTP) [11] is used to transfer data. The window scaling option as specified by RFC1323 [3] is set to 4, allowing the receiver to advertise a window up to 1048560 bytes. The receiver socket buffer size was set to 400 Kbytes, and 41 Mbytes were transferred in each test. TCP traces are collected using the *trcollect* utility, provided on Cray systems.

We present our measurements in the next section, along with a discussion of the implications of each set of observations.

## 7 Analysis

In this section we examine the effect of blocking factor and gateway hold time on packet loss and delay using an unreliable protocol (ICMP/IP). We then turn to the effects of HIPPI blocking on a TCP/IP connection.

### 7.1 Gateway hold time $h = \infty$

Figure 3 shows the mean packet **delay** on the blocked connection as a function of blocking factor, for several source packet rates and sizes. When $\lambda < \mu$, mean delay increases with blocking factor. When $h = \infty$, a remote HSG never discards packets but instead allows queues to build. The mean delay thus increases with the blocking factor. Furthermore, mean delay is insensitive to packet size because per packet connection wait time dominates the per-byte transmission time. When $\lambda > \mu$, mean delay grows unbounded (ultimately resulting in source HSG flow control), as indicated by the vertical arrows. For example, in the 100 pps cases, this occurs at a blocking factor of 0.01.

Figure 4 shows the measured **packet loss** percentage corresponding to the delay graph Figure 3. When $h = \infty$, packet loss only occurs when the HSG burst buffers overflow. Observed packet loss is due to packet discards at a hardware flow–controlled sender which does not employ higher–layer flow control (in our case ICMP/IP as used by *ping*). When $h = \infty$, an HSG with full burst buffers will signal the remote HSG to stop accepting new connections. Loss remains constant at 0 except when, for a fixed blocking factor, the source $rate * size$ product is large enough to overflow the burst buffers (the *saturation point*). For example, in the 64 Kbyte-100 pps case, the saturation point occurs for $0.005 \leq BF < 0.01$. Beyond the saturation point (provided $\lambda < \mu$), packet loss grows exponentially with the blocking factor for a fixed $rate * size$ product.

### 7.2 Gateway hold parameter $h = 10ms$

Figure 5 recreates the tests performed in Section 7.1 but with the gateway hold parameter $h = 10ms$. In this case, an HSG unable to establish a connection in 10ms will discard queued traffic for the pending connection. Thus,

scheme is limited to approximately 10 km distance, beyond which HIPPI signal regenerators are required. The fiber-extender is scheduled to be replaced with a pair of HSGs.

the mean RTT of a blocked (but successfully delivered) packet is bounded by $h + (unblocked)RTT$. This effect is illustrated by the increase in mean delay for $BF < h$. When $BF \geq h$, we observe a horizontal trend toward the $h + (unblocked)RTT$ bound. In our case, the mean unblocked RTT ranges between 4ms and 6ms for packets sized 64bytes and 64Kbytes, respectively.

Packet loss for the $h = 10ms$ experiment is illustrated in Figure 6. We observe loss to be largely insensitive to both packet size and rate for a fixed blocking factor. When $BF < h$ (below 0.01), packet discards occur exactly when the instantaneous blocking time exceeds $h$. Because the blocking connection's hold time is exponentially distributed, its standard deviation is equal to its mean, and thus aggregate packet loss increases with proximity to $h$. For a fixed $BF$, each instance of instantaneous blocking time exceeding $h$ causes packet loss. In comparison to the $h = \infty$ case, this case results in a higher aggregate packet loss. When $BF > h$, only those packets encountering an instantaneous blocking time below $h$ will be successfully delivered. As $BF$ increases the probability of successful delivery decreases until no packets can be delivered.

## 7.3 TCP

We now turn our attention to the effect of packet loss and delay on TCP in our experimental environment. Figures 7 and 8 show the sequence number and congestion window versus time for three distinct TCP conversations. Without blocking (and without the 100pps limitation from sections 7.1 and 7.2), we measured an upper bound on application throughput of 314.4Mb/s.

For the blocked cases, $BF$ is fixed at 0.02 seconds for the two values of $h$ used previously. The blocking connection, initiated by the Sun workstation, keeps the receive port of the C90 busy. During a unidirectional file transfer from the C90 to the YMP, ACK packets returning from the YMP to the C90 may be lost or delayed.

When $h = \infty$, ACKs are delayed but not lost, causing throughput degradation. The measured throughput in this case is 76.8 Mb/s, only 24.4% of the best case. The sender's congestion window ramps up to the receiver's advertised window and remains unaffected by the delayed ACKs, as illustrated in Figure 8.

When $h = 10ms$, the greater ACK loss rate causes the sender's retransmission timer to expire, inducing slow-start behavior. The measured throughput in this case is 10.4 Mb/s, or 3.3% of the best case. This is illustrated in Figure 7 by observing that segments are retransmitted and in Figure 8 by the sawtooth character of the congestion window plot, in contrast to the congestion window seen in the no-blocking and $h = \infty$ cases.

## 8 Conclusions

In this study, we have verified our hypothesis that HIPPI blocking behavior occurs and has a measurable effect on packet loss and delay. We have also observed the measured character of packet loss and delay is associated with the gateway hold time parameter $h$ as expected. In the presence of blocking, there exists a tradeoff between larger round-trip delays (and variance) versus greater packet loss. For infinite $h$, packet loss is reduced at the expense of increased delay (and variance), in contrast with finite $h$ which provides bounded delays and queue occupancies at the expense of increased packet loss.

When $h = \infty$, the CASA HIPPI WAN resembles a HIPPI LAN. In a local HIPPI environment, the switching fabric will not discard frames because of congestion. Similarly, in the CASA network when $h = \infty$, flow control preserves local HIPPI reliability semantics by instituting admission control during saturation periods (in preference to dropping packets). We therefore anticipate all CASA findings with $h = \infty$ to apply as well to HIPPI LANs.

The delay–vs–loss tradeoff affords the opportunity to match network characteristics with application/transport requirements. With respect to TCP, high packet loss induces the slow-start policy even when the network may not be congested, leading to low throughput. In addition, the variance-sensitive TCP RTT estimator is sufficiently robust to avoid spurious retransmissions in HIPPI–based networks such as CASA. Thus, the delay/loss tradeoff for TCP clearly favors limiting packet loss. Other transport protocols or loss–tolerant applications may prefer a less reliable channel in exchange for bounded delay variance.

In a general purpose network, applications should be permitted to specify a preference with respect to the tradeoff mentioned above by indicating a desired type of service. The HSG presently lacks support for setting $h$ based on type-of-service. Furthermore, supporting type of service requires protocol-level identification of an application's preferences, and is not supported by HIPPI. Providing router functionality within the HSG and employing a network layer protocol capable of specifying type of service handling would allow HIPPI LANs to be effectively extended to the WAN environment.

## References

[1] Jacobson, V., "Congestion Avoidance and Control", Proc. ACM SIGCOMM 1988, Stanford, CA, Aug 1988.

[2] Postel, J., "Internet Protocol", RFC 791, Sep 1981.

[3] Jacobson, V., Braden, R., Borman, D., "TCP Extensions for High Performance", RFC 1323, May 1992.

[4] Chlamtac, I., Ganz, A. and Kienzle, M., "An HIPPI Interconnection System", IEEE Trans. Comput., vol 42, pp. 138–149.

[5] P. Messina, "CASA Gigabit Network Testbed", Proc. Supercomputing '91, November 1991

[6] ANSI X3T9.3, "High-Performance Parallel Interface:HIPPI-PH, HIPPI-SC, HIPPI-FP and HIPPI-LE", Amer. Nat'l. Std. for Info. Sys., Aug 1993.

[7] Network Systems Corporation, *HIPPI Connectivity Solutions*, 1993

[8] Moore, R. "Distributing Applications Across Wide Area Networks", General Atomics Technical Report GA-A20074, May 1990.

[9] St. John, W., *Personal Communication*, Feb. 1994.

[10] Shenker, S., Zhang, L., "Some Observations on the Dynamics of a Congestion Control Algorithm", Computer Communications Review, Vol. 20, no. 5, Oct. 1990.

[11] J. Postel, J. Reynolds, "File Transfer Protocol", RFC 959, Jan 1985.

[12] Ballart, R., Ching Y., "SONET: Now it's the Standard Optical Network", IEEE Communications Magazine, Vol. 29, no. 3, Mar. 1989.

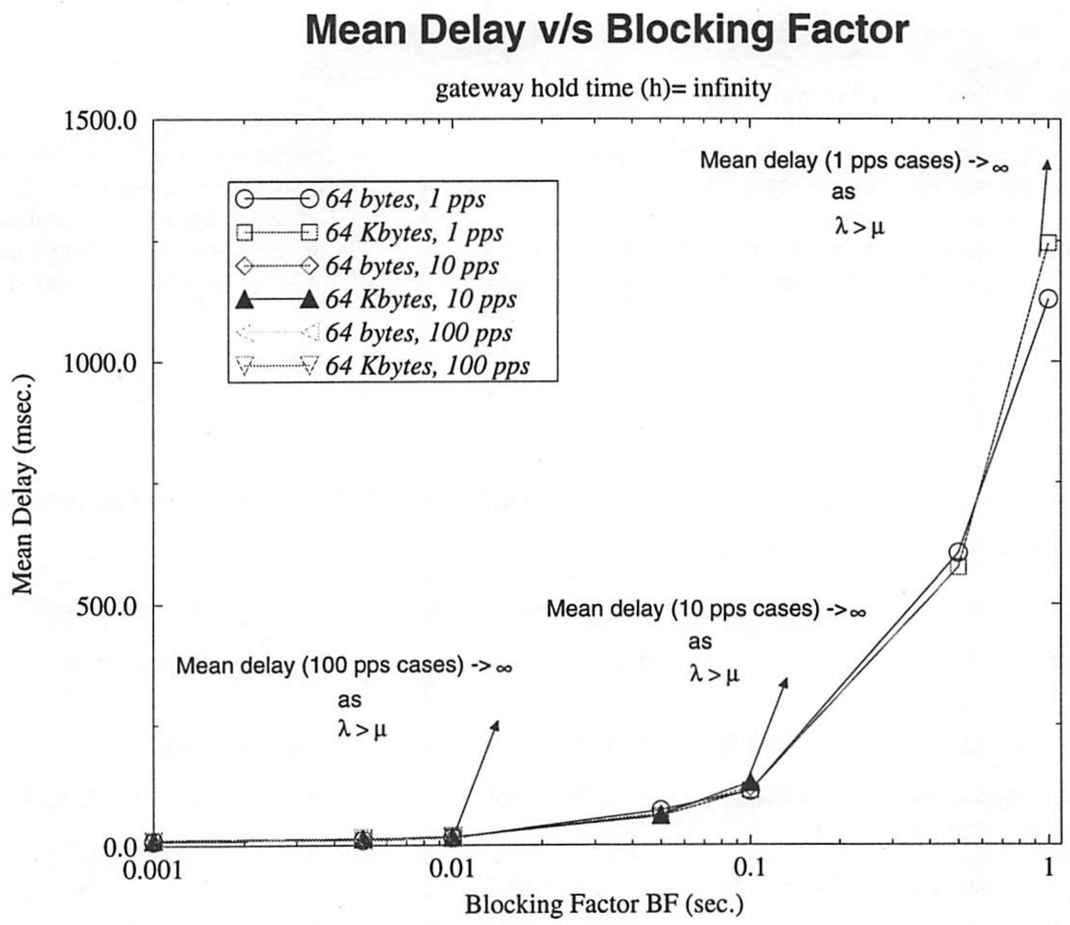


Figure 3: Mean Packet Delay vs Blocking Factor for $h = \infty$

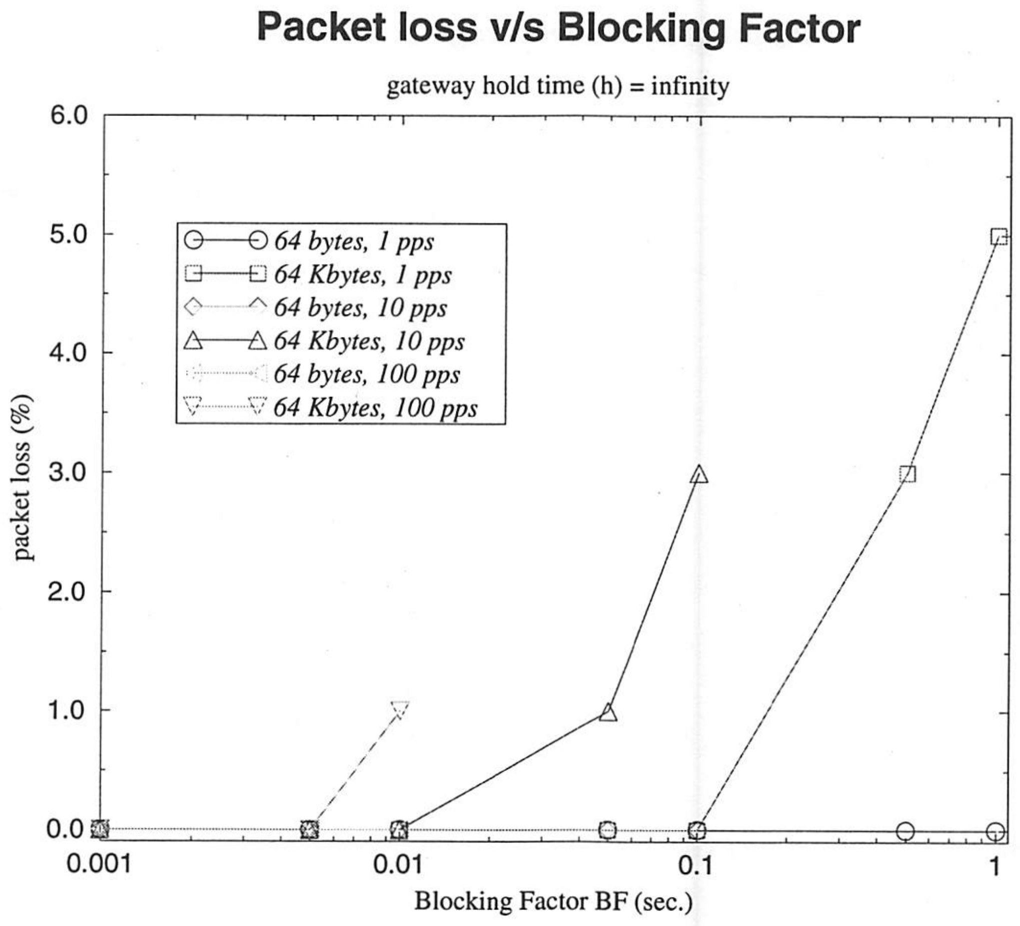


Figure 4: Packet Loss vs Blocking Factor for $h = \infty$

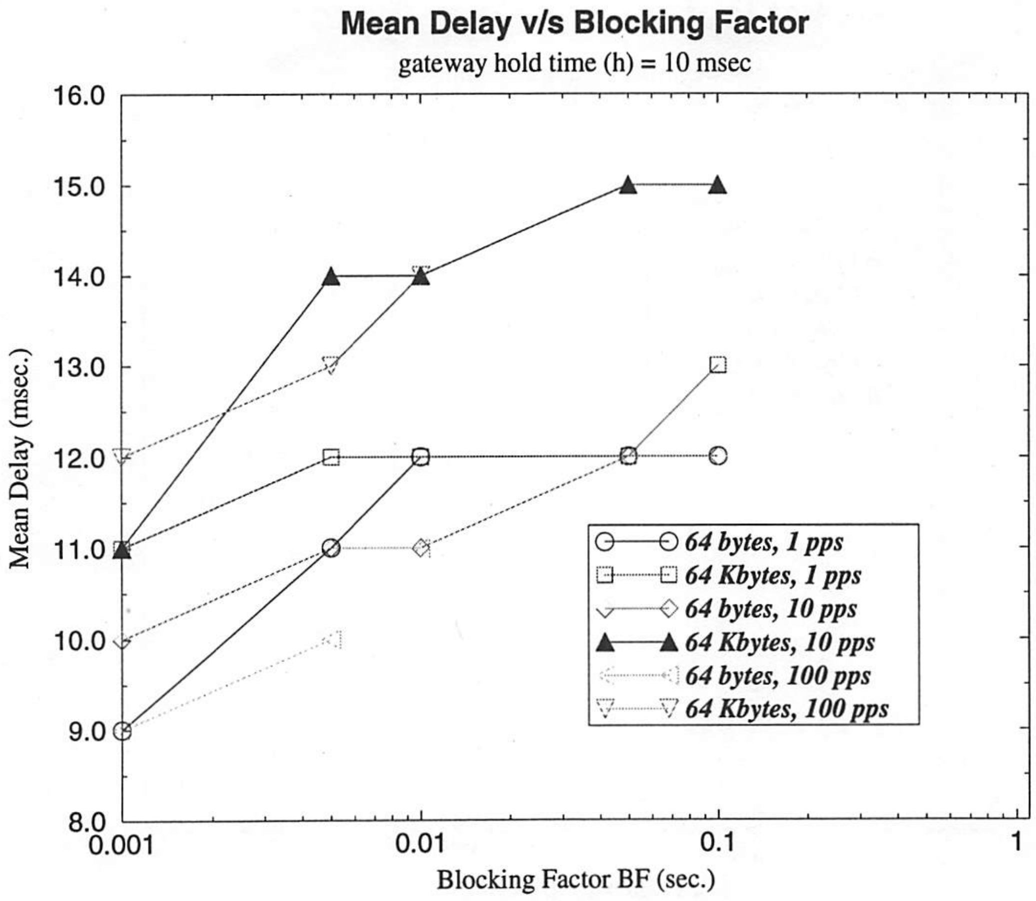


Figure 5: Mean Packet Delay vs Blocking Factor for $h = 10ms$

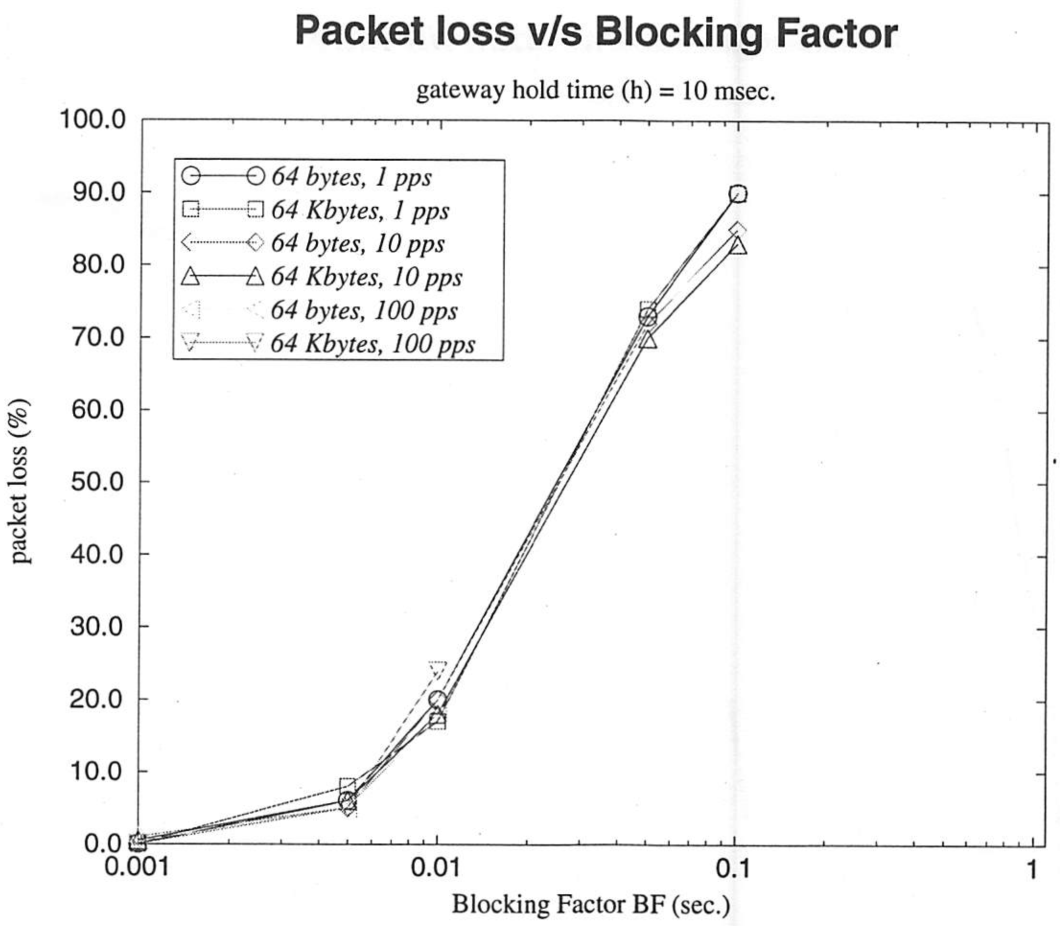


Figure 6: Packet Loss vs Blocking Factor for $h = 10ms$

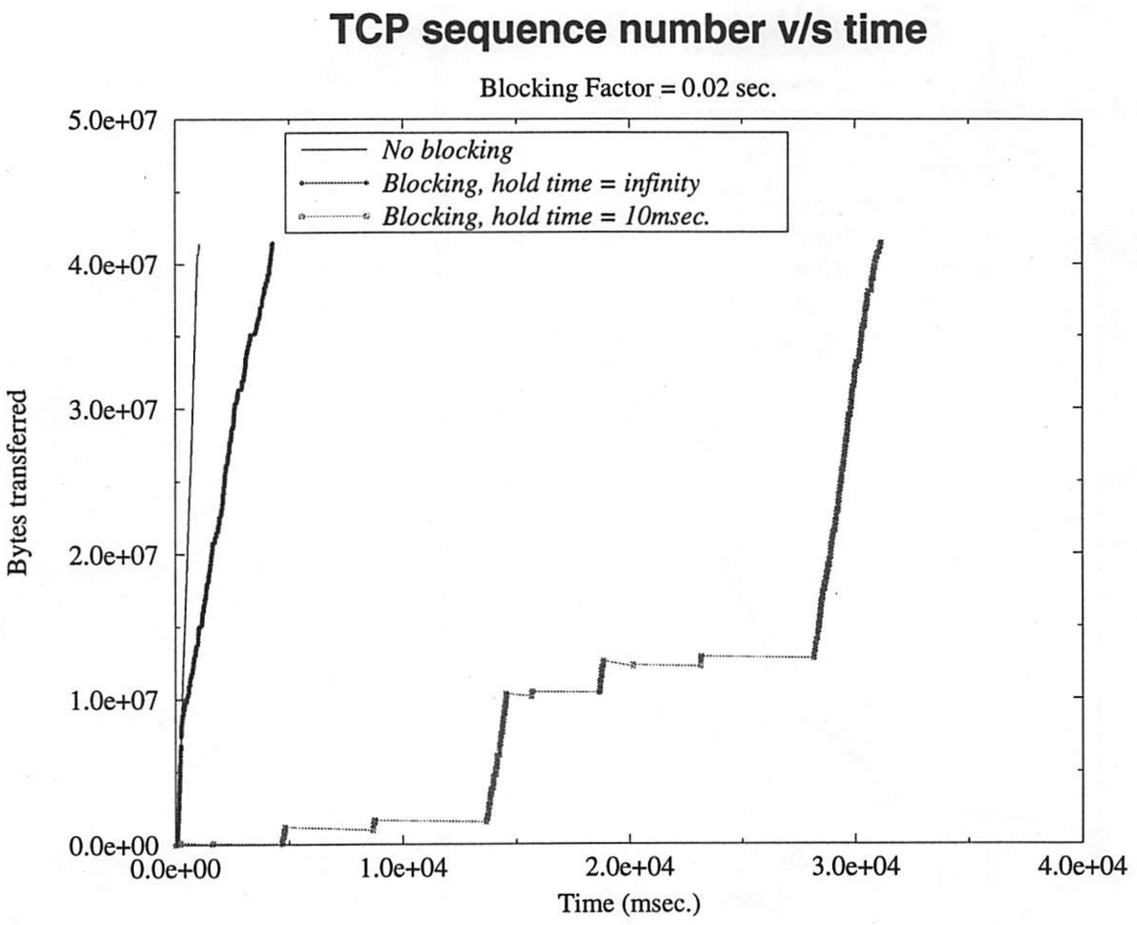


Figure 7: TCP Sequence Number vs Time

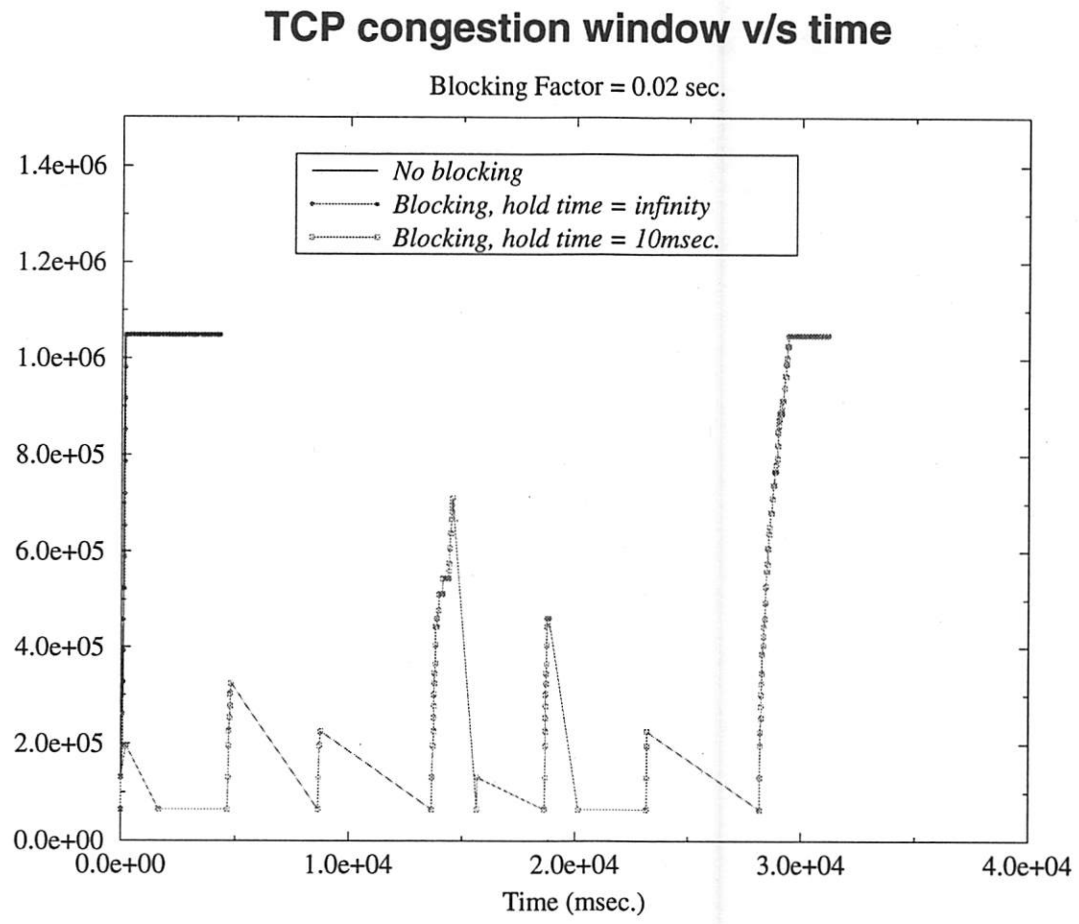


Figure 8: TCP Congestion Window vs Time

# System Issues in Implementing High Speed Distributed Parallel Storage Systems

*Brian Tierney (bltierney@lbl.gov),*
*Bill Johnston[1] (wejohnston@lbl.gov),*
*Hanan Herzog, Gary Hoo, Guojun Jin, Jason Lee*

*Imaging Technologies Group*
*Lawrence Berkeley Laboratory[2]*
*Berkeley, CA 94720*

## Abstract

In this paper we describe several aspects of implementing a high speed network-based distributed application. We describe the design and implementation of a distributed parallel storage system that uses high speed ATM networks as a key element of the architecture. The architecture provides what amounts to a collection of network-based disk block servers, and an associated name server that provides some file system functionality. The implementation approach is that of user level software that runs on UNIX workstations. Both the architecture and the implementation are intended to provide for easy and economical scalability in both performance and capacity. We describe the software architecture, the implementation and operating system overhead issues, and our experiences with this approach in an IP-over-ATM WAN. Although most of the paper is specific to a distributed parallel data server, we believe many of the issues we encountered are generally applicable to any high speed network-based application.

## 1.0 Introduction

In recent years many technological advances have made distributed multimedia servers a reality, and people now desire to put "on-line" large amounts of information, including images, videos, and hypermedia databases. Increasingly, there are applications that demand high-bandwidth access to this data, either in single user streams (e.g., large image browsing, uncompressible scientific and medical video, and multiple coordinated multimedia streams) or, more commonly, in aggregate for multiple users. Here we describe a network distributed data server, called the Image Server.System (ISS). The ISS is being used to supply data to a terrain visualization application that requires 300-400 Mbits/s of data to provide a realistic visualization. Both the ISS and the application have been developed in the context of the MAGIC Gigabit testbed ([5] and [11]).

1. Correspondence should be directed to Bill Johnston, Lawrence Berkeley Laboratory, Bldg. 50B - 2239, Berkeley, CA, 94720. Tel: 510-486-5014, fax: 510-486-6363).

2. This work is jointly supported by ARPA - CSTO, and by the U. S. Dept. of Energy, Energy Research Division, Office of Scientific Computing, under contract DE-AC03-76SF00098 with the University of California. This document is LBL report LBL-35775.

To address a point frequently raised, compression is not practical in the case of terrain visualization because the cost of decompression is prohibitive in the absence of suitable hardware implementations.

To comment briefly on the relevant technologies, current disk technology delivers about 4 Mbytes/s (32 Mbits/s), a rate that has improved at about 7% each year since 1980 [10], and there is reason to believe that it will be some time before a single disk is capable of delivering streams at the rates needed for the application mentioned. While RAID [10] and other parallel disk array technologies can deliver higher throughput, they are still relatively expensive, and do not scale well (at least economically), especially in the scenario of multiple network-distributed users (where we assume that the sources of data, as well as the multiple users, will be widely distributed). Wide area Asynchronous Transfer Mode (ATM) networks are being built on a SONET infrastructure, which has the characteristic that bandwidth aggregates upward (contrary to our current network hierarchy, where the slowest networks are at the top of the hierarchy). This characteristic makes it possible to use the network to aggregate many low-speed sources into a single high-speed stream.

The approach described here differs in many ways from RAID, and should not be confused with it. RAID uses a particular data layout and redundancy strategy to secure reliable data storage and parallel disk operation. Our approach, while using parallel disks and servers, imposes no particular layout strategy (in fact, this is deliberately left to the application domain), and is implemented entirely in software (though the data redundancy idea of RAID might be applied across servers).

At the present state of development and experience, the system we describe is used primarily as a large, fast "cache" for image or multimedia data. In our approach, reliability with respect to data corruption is provided by the usual OS and disk controller-level mechanisms, and reliability of the overall system is a function of user-level strategies of data replication. The data of interest (tens to hundreds of GBytes) is typically loaded from archival tertiary storage, or written into the system from live video sources. (In the latter case, the data is also archived to bulk storage in real-time.)

The Image Server System is an example of distributed parallel data storage technology. It is a multimedia file server that is distributed across a wide area network to supply data to applications located anywhere in the network. This system is not a general purpose, distributed file system in that the data units ("files") are assumed to be large and relatively static. The approach allows data organization to be determined by the user as a function of data type and access patterns. For most applications, the goal of data organization is that data should be declustered across both disks and servers (that is, dispersed in such a way that as many system elements as possible can operate simultaneously to satisfy a given request). This declustering allows a large collection of disks to seek in parallel, and allows all servers to send the resulting data to the application in parallel, enabling the ISS to perform as a high-speed image server. Architecturally, the ISS is a distributed, parallel mass storage system that uses UNIX workstation technology to provide a low-cost, scalable implementation. The data transport is via TCP/IP or RTP/IP[13]. The scalability arises from the high degree of independence among the servers: both performance and capacity may be increased, in essentially linear fashion, by adding servers. (Ultimately this is limited by the parallelism inherent in the data.) The general idea is illustrated in Figure 1 (Data Streams Aggregated by ATM Switches).

In our prototypes, the typical ISS consists of several (four - five) UNIX workstations (e.g. Sun SPARCStation, DEC Alpha, SGI Indigo, etc.), each with several (four - six) fast-SCSI disks on multiple (two - three) SCSI host adaptors. Each workstation is also equipped with an ATM network interface. An ISS configuration such as this can deliver an aggregated data stream to an application at about 400 Mbits/s (50 Mbytes/s) using these relatively low-cost, "off the shelf" components by exploiting the parallelism provided by approximately five hosts, twenty disks, ten SCSI host adaptors, and five network interfaces.

## 2.0 Related Work

In some respects, the ISS resembles the Zebra network file system, developed by John H. Hartman and John K. Ousterhout at the University of California, Berkeley [4]. Both the ISS and Zebra can separate their file access and management activities across several hosts on a network. Both try to maintain the availability of the system as a whole by building in some redundancy, allowing for the possibility that a disk or host might

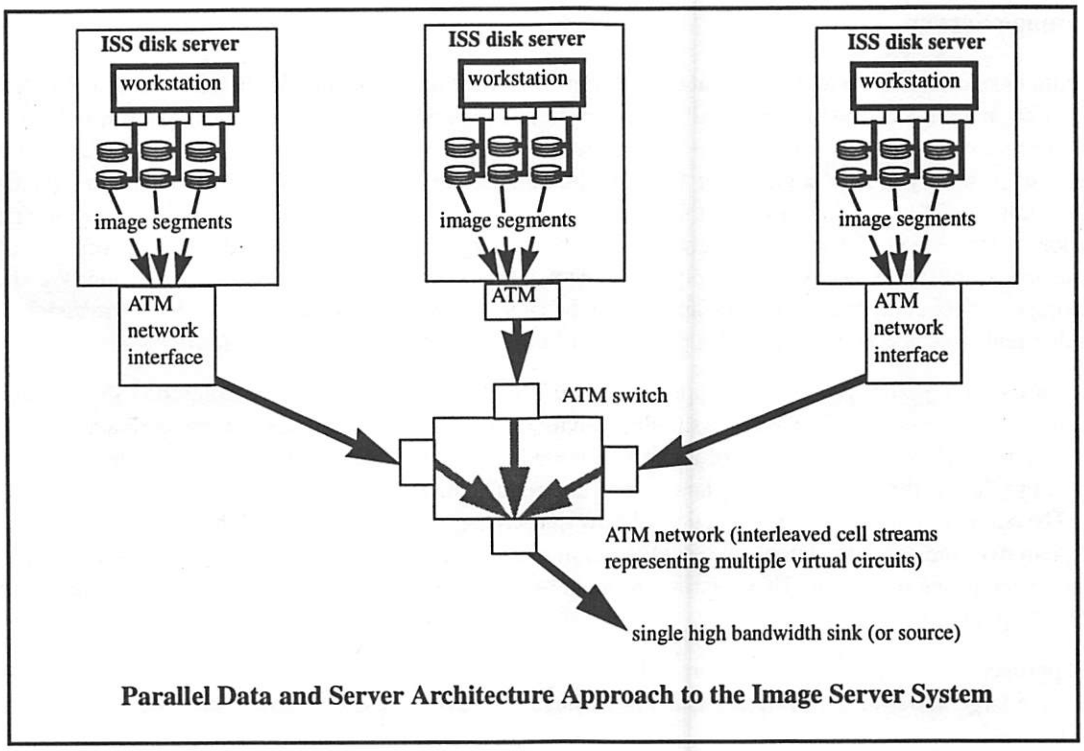

ISS disk server
workstation
image segments
ATM network interface
ISS disk server
workstation
image segments
ATM
ISS disk server
workstation
image segments
ATM network interface
ATM switch
ATM network (interleaved cell streams representing multiple virtual circuits)
single high bandwidth sink (or source)
Parallel Data and Server Architecture Approach to the Image Server System

**Figure 1 Data Streams Aggregated by ATM Switches**

be unavailable at a critical time. The goal of both is to increase data throughput despite the current limits on both disk and host throughput.

However, the ISS and the Zebra network file system differ in the fundamental nature of the tasks they perform. Zebra is intended to provide traditional file system functionality, ensuring the consistency and correctness of a file system whose contents are changing from moment to moment. The ISS, on the other hand, tries to provide extremely high-speed, high-throughput access to a relatively static set of data. It is optimized to retrieve data, requiring only minimum overhead to verify data correctness and no overhead to compensate for corrupted data.

There are other research groups working on solving problems related to distributed storage and fast multimedia data retrieval. For example, Ghandeharizadeh, Ramos, et al., at USC are working on declustering methods for multimedia data [3], and Rowe, et al., at UCB are working on a continuous media player based on the MPEG standard [12].

## 3.0 Applications

There are several target applications for the initial implementation of the ISS. These applications fall into two categories: image servers and multimedia / video file servers.

### 3.1 Image Server

The initial use of the ISS is to provide data to a terrain visualization application in the MAGIC testbed[3]. This application, known as TerraVision [5], allows a user to navigate through and over a high resolution landscape represented by digital aerial images and elevation models. TerraVision is of interest to the U.S. Army because of its ability to let a commander "see" the battlefield. TerraVision is very different from a typical "flight simulator"-like program in that it uses large quantities of high resolution aerial imagery for the visualization instead of simulated terrain. TerraVision requires large amounts of data, transferred at both bursty and steady rates. The ISS is used to supply image data at hundreds of Mbits/s rates to TerraVision. We are not using any data compression for this application because the bandwidth requirements for TerraVision are such that real-time decompression is not possible without using special purpose hardware.

In the case of a large-image browsing application like TerraVision, the strategy for using the ISS is straightforward: the image is tiled (broken into smaller, equal-sized pieces), and the tiles are scattered across the disks and hosts of the ISS. The order of images delivered to the application is determined by the application predicting a "path" through the image (landscape), and requesting the tiles needed to supply a view along the path. The actual delivery order is a function of how quickly a given server can read the tiles from disk and send them over the network. Tiles will be delivered in roughly the requested order, but small variations from the requested order will occur. These variations must be accommodated by buffering or other strategies in the client application.

This approach can supply data to any sort of large-image browsing application, including applications for displaying large aerial-photo landscapes, satellite images, X-ray images, scanning microscope images, and so forth.

### 3.2 Video Server

Examples of video server applications include video players, video editors, and multimedia document browsers. A video server might contain several types of stream-like data, including conventional video, compressed video, variable time base, multimedia hypertext, interactive video, and others. Several users may be accessing the same video data at the same time, but may be at different frames in the stream. This affects many factors in an image server system, including the layout of the data on disks. Since there is a predictable, sequential pattern to the requests for data, the data would be placed on disk in a like manner. Large commercial concerns such as Time Warner and U.S. West are building large-scale commercial video servers such as the Time Warner / Silicon Graphics video server [6]. Our approach may address a wider scale, as well as a greater diversity, of data organization strategies so as to serve the diverse needs of schools, research institutions, and hospitals for video-image servers in support of various educational and research-oriented digital libraries.

### 3.3 Application to ISS Interface

Application access to the ISS is through a client-side library that accepts requests for data, and returns data to the application. The client library obtains from the ISS Master a list of ISS Disk Servers (q.v.) that have data for the area of interest. The client library establishes connections to all ISS Disk Servers containing that data set. The application specifies the location of a memory buffer to receive incoming data.

The current implementation provides access to large images, in which the unit of data is a tile. The application requests data in terms of lists of tiles, and the tiles sent by the ISS servers are placed into the application's buffer. (See Figure 2 (Application Architecture).)

3. MAGIC (Multidimensional Applications and Gigabit Internetwork Consortium) is a gigabit network testbed that was established in June 1992 by the U. S. Government's Advanced Research Projects Agency (ARPA)[11]. MAGIC's charter is to develop a high-speed, wide-area networking testbed that will demonstrate interactive exchange of data at gigabit-per-second rates among multiple distributed servers and clients using a terrain visualization application.

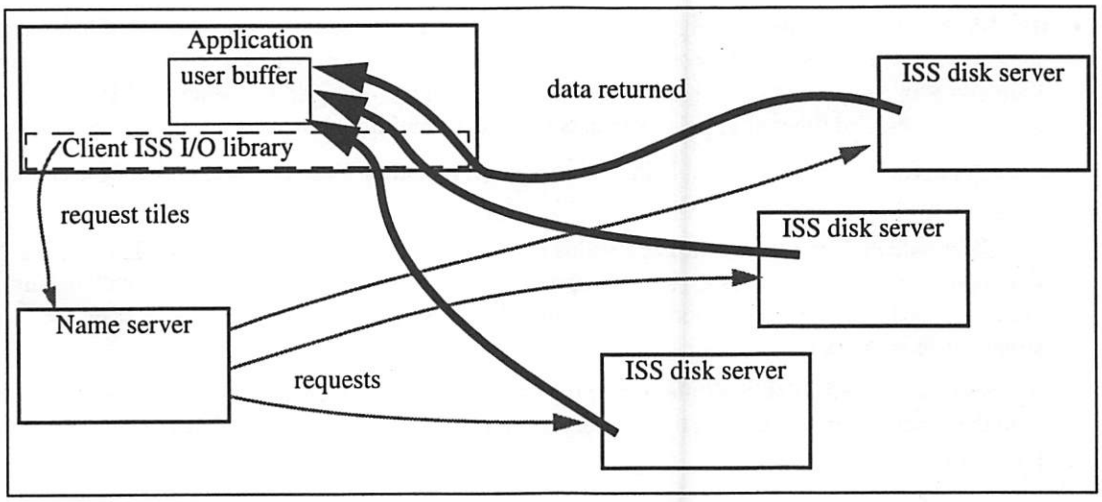


**Figure 2 Application Architecture**

### 3.4 Data Prediction

Data prediction is important to ensure that the ISS is utilized as efficiently as possible. By always requesting more tiles than the ISS can actually deliver before the next tile request, we ensure that no component of the ISS is ever idle. For example, if most of a request list's images were on one server, the other servers could still be reading and sending or caching images that may be needed in the future, instead of idly waiting. The point of the path prediction is to provide a rational basis for pre-requesting tiles.

As an example of path prediction, consider an interactive video database with a finite number of possible branch points. (A "branch point" occurs where a user might select one of several possible play clips.) As a branch point is approached by the player, the predictor (without knowledge of which branch will be taken) will start requesting images (frames) along both branches. These images are cached first at the disk servers, then at the receiving application. As soon as a branch is chosen, the predictor ceases to send requests for images from the other branch. Any cached but unsent images are flushed as better predictions fill the cache.

For MAGIC's TerraVision, prediction is based on geometric characteristics of the path being followed, the limitations of the mode of simulated transport (that is, walking, driving, flying, etc.), the intended destination, and so on. The prediction results in a priority ordered list of tile requests being sent to the ISS. The ISS has no knowledge of the prediction strategy (or even if one has been used).

The client will keep asking for an image until it shows up, or until it is no longer needed (e.g. the application may have "passed" the region of landscape that involves the image that was requested, but never received.) Applications will have different strategies to deal with images that do not arrive in time. For example, TerraVision keeps a local low resolution data set to fill in for missing tiles.

Prediction is transparent to the ISS, and is manifested only in the order and priority of images in the request list. The prediction algorithm is mostly a function of the client application, and typically runs on the client. Prediction could also be done by a third-party application. The aforementioned interactive video database, for example, might use a third-party application for prediction.

## 4.0 Design

### 4.1 Goals

The following are some guidelines we have followed in designing the ISS:

- The ISS should be capable of being geographically distributed. In a future environment of large-scale, mesh-connected national networks, network-distributed storage should be capable of providing an uninterruptable stream of data, in much the same way that a power grid is resilient in the face of source failures, and tolerant of peak demands because of multiple sources multiply interconnected.
- The ISS approach should be scalable in all dimensions, including data set size, number of users, number of server sites, and aggregate data delivery speed.
- The ISS should deliver coherent image streams to an application, given that the individual images that make up the stream are scattered (by design) all over the network. In this case, "coherent" means "in the order needed by the application." No one disk server will ever be capable of delivering the entire stream. The network is the *server.*
- The ISS should be affordable. While something like a HIPPI-based RAID device might be able to provide the functionality of the ISS, this sort of device is very expensive, is not scalable, and is a single point of failure.

## 4.2 Research Issues

The design goals present several issues that need to be addressed. These include:

- How to store and retrieve image data at gigabit speeds using a storage system whose servers are widely distributed;
- How to place data blocks (tiles) such that image data will be distributed across many storage servers in a way that ensures parallel operation;
- How to handle high-speed IP transport over ATM networks to provide the parallel data paths needed to aggregate medium-speed disk servers into a logically integrated, high-speed image storage server. (Although ATM will probably become the Ethernet of the future, end-to-end networks will be heterogeneous for a long time to come, necessitating the use of an internetwork protocol, of which IP is the clear choice);
- Assessing how an ATM network will behave (or misbehave) under the conditions of multiple, coordinated, parallel data streams.

## 4.3 Approach: A Distributed, Parallel Server

The ISS design is based on the use of multiple low-cost, medium-speed disk servers which use the network to aggregate server output into a single high-bandwidth stream. To achieve high performance we exploit all possible levels of parallelism, including that available at the level of the disks, controllers, processors/memory banks, servers, and the network. Proper data placement strategy is also key to exploiting system parallelism. For placement of image tile data, an application-related program declusters tiles so that all ISS disks are evenly accessed by tile requests, but clusters tiles that are on the same disk[1]. This strategy is a function of both the data structure (tiled images) and the geometry of the access (e.g., paths through the landscape). Currently we are working on extending these methods to handle video-like data.

At the individual server level, the approach is that of a collection of disk managers that move requested data to memory cache. Depending on the nature of the data and its organization, the disk managers may have a strategy for moving other closely located and related data from disk to memory as a side effect of a particular data request. However, in general, we have tried to keep the implementation of data prediction (determining what data will be needed in the near future) separate from the basic data moving function of the server. Prediction might be done by the application, or it might be done be a third party that understands the data usage patterns. In any event, the server sees only lists of requested blocks.

As explained below, the dominant bottlenecks for this type of application in a typical UNIX workstation are, first, memory copy speed, and second, network access speed. For these reasons, an important design criterion is to use as few memory copies as possible, and to keep the network interface operating at full bandwidth all the time. Our implementation uses only three copies to get data from disk to network, so maximum server throughput is about (mem_copy_speed / 3).

Another important aspect of the design is that all components are instrumented for timing and data flow monitoring in order to characterize the ISS implementation and the network performance. To do this, all communications between ISS components are timestamped. We are using GPS (Global Positioning System) receivers and NTP (Network Time Protocol) [8] [9] to synchronize the clocks of all ISS servers and of the client application in order to accurately measure network throughput and latency.

## 5.0 ISS Architecture and Implementation

The following is a brief overview of a typical ISS operation. A data set must first be loaded across a given set of ISS hosts and disks, and a table containing disk/offset locations for each block of data is stored on each host. The application sends requests for data (images, video, sound, etc.) to the Name Server process on each Disk Server host, which does a lookup to determine the location (server - disk - offset) of the requested data. If the data is not stored on that host, the request is discarded with the assumption that another host will handle it; otherwise the list of locations is passed to the ISS Disk Server. Each Disk Server then checks to see if the data is already in its cache, and if not, fetches the data from disk and transfers it to the cache. Once the data is in the cache, it is sent to the requesting application.

In the following sections, we describe the basic software modules, their functions, how they relate to each other, and some of the terms and models that were used in the design of the ISS. Figure 3 (ISS Server Archi-

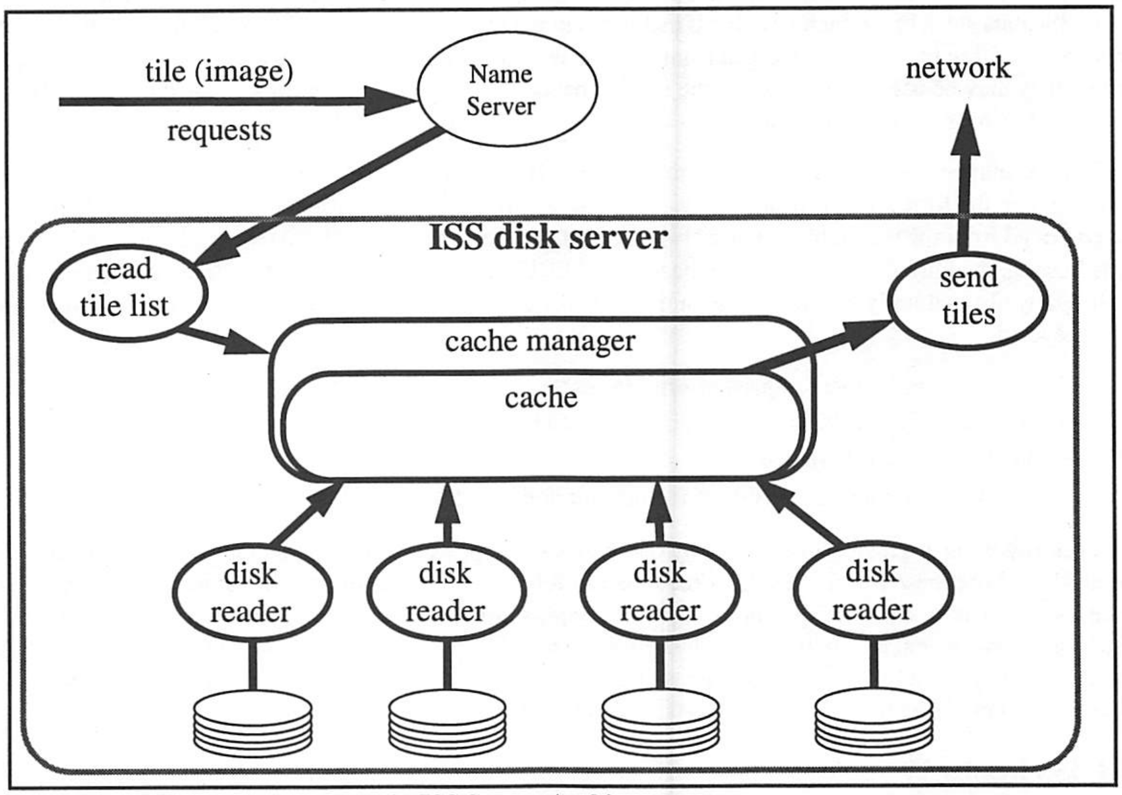


**Figure 3 ISS Server Architecture**

tecture) shows how the components of the ISS relate to each other.

## 5.1 ISS Master

The ISS Master process is responsible for application-to-ISS startup communication, Disk Server process initialization, performance monitoring, and coordination between multiple ISS Disk Servers. This includes the ability to collect performance and usage statistics of all ISS components. In the future, we plan to extend the functionality of the Master to dynamically reconfigure ISS Disk Server usage to avoid network or ISS Disk Server bottlenecks.

## 5.2 Name Server

The Name Server listens for tile request lists from the application. After receiving a list, the Name Server does a table lookup to determine where the data is located (i.e. which server, which disk, and the disk offset). The Name Server then passes this list to the ISS Disk Server.

## 5.3 ISS Disk Server

There is one ISS Disk Server process for each ISS host. It is responsible for all ISS memory buffer and request list management on that host. The Disk Server receives image requests from the Master process, and determines if the image is already in its buffer cache. If it is already in the buffer cache (which is kept entirely in available memory), the request is added to the "to send" list. Otherwise, it is added to a "to read" list. Tile requests that have not been satisfied by the time the next list from the Master process arrives are "flushed" (discarded) from the lists. All requests that haven't been either read off of disk or written to the network interface are removed from all request lists, and any memory buffers waiting to be written are returned to the hash table. Note that if a tile read has completed, but the tile has not yet been sent to the network, the data stays in the cache, so that if that tile is in the next request list it will be sent first. Those buffers that were waiting to be filled with data from the disk are put at the head of an LRU (Least Recently Used) list so they may be used for requests in the newly arrived list. The Disk Server process also periodically sends status information to the Master.

ISS buffer management is very similar to that of the UNIX operating system, and many of the ideas for lists, hashing, and the format of the headers have been adopted from UNIX for use within the ISS [7]. A buffer can be freed from the hash table in one of two ways. If a buffer was allocated to a list (read/send) and that list was flushed, the buffer is returned to the head of the LRU list so that it is the next buffer to be reused. A buffer may also naturally progress through the LRU list until it has reached the end of the list, at which time it is recycled.

The Disk Server handles three request priority levels:

- high: send first, with an implicit priority given by order within the list.
- medium: send if there is time.
- low: fetch into the cache if there is time, but don't send.

The priority of a particular request is set by the requesting application. The application's prediction algorithm should use these priority levels to keep the ISS fully utilized at all times without requesting more data than the application can process. For example, the application should send low priority requests to pull data into the ISS cache that the application will need in the near future; this data is not sent to the application until the application is ready. Another example is an application that plays back a movie with a sound track, where audio might be high priority requests, and video medium priority requests.

## 5.4 ISS Reader

The ISS Reader process reads data off of disk and puts it into the buffer cache that is managed by the Disk Server process. There is one Reader per physical disk. This process continually checks for requests in the "to read" list, starts a read operation on that disk if a request is pending, then waits for the data to be read. Once the data is read off of disk the request is moved into the list of data that is to be written out. There are two distinct lists of data that are to be written out, one for each of the high and medium priority levels described above.

## 5.5 ISS Sender

The ISS Sender process sends all data in the "to send" list out to the application that requested it. There is one sender per network interface. This process continually checks the list of data that is ready to be written out, looking for data that is of high or medium priority (as described above). Note that data of medium priority will only be sent if there is no data of high priority in the cache. However, it is possible for medium priority data to be written out before higher priority data, as in the case where the medium priority data is in the memory cache, and higher priority data is resident on disk.

## 6.0 Current ISS Status

All ISS software is currently tested and running on Sun workstations (SPARCstations and SPARCserver 1000's) running SunOS 4.1.3 and Solaris 2.3, DEC Alpha's running OSF/1, and SGI's running IRIX 5.x. Demonstrations of the ISS with the MAGIC Terrain Visualization application TerraVision have been done using several WAN configurations in the MAGIC testbed [11]. Using enough disks (4-8, depending on the disk and system type), the ISS software has no difficulty saturating current ATM interface cards. We have worked with 100 Mbit and 140Mbit TAXI S-Bus and VME cards from Fore systems, and OC-3 cards from DEC, and in all cases ISS throughput is only slightly less than *ttcp*[4] speeds.

Table 1 below shows various system *ttcp* speeds and ISS speeds. The first column is the maximum *ttcp*

**TABLE 1.**

| System | Max ATM LAN *ttcp* | *ttcp* w/ disk read | Max ISS speed |
|---|---|---|---|
| Sun SS10-51 | 70 Mbits/sec | 60 Mbits/sec | 55 Mbits/sec |
| Sun SS1000 (2 proc) | 75 Mbits/sec | 65 Mbits/sec | 60 Mbits/sec |
| SGI Challenge L | 82 Mbits/sec | 72 Mbits/sec | 65 Mbits/sec |
| Dec Alpha | 127 Mbits/sec | 95 Mbits/sec | 88 Mbits/sec |

speeds using TCP over a ATM LAN with a large TCP window size. In this case, *ttcp* just copies data from memory to the network. For the values in the second column, we ran a program that continuously reads from all ISS disks simultaneously with *ttcp* operation. This gives us a much more realistic value for what network speeds the system is capable or while the ISS is running. The last column is the actual throughput values measured from the ISS. These speeds indicate that the ISS software adds a relatively small overhead in terms of maximum throughput.

## 7.0 Operation System Issues

### 7.1 Threads

Currently, the ISS Disk Server is implemented as a group of loosely-coordinated UNIX processes. We believe performance can be enhanced by transforming these processes into threads. Most of the gains arise from bypassing the overhead of the interprocess communication mechanisms needed to guarantee consistency of resources shared by the processes, e.g., the semaphores needed to ensure non-simultaneous access to the to-read and to-send lists. The same functionality can be achieved using thread-based mechanisms that are designed to be much faster, e.g., mutual exclusion locks.

The ISS Disk Server requires separate processes to receive from the network, read from disk, and send to the network. These processes must share certain resources, namely, the to-read lists, the to-send list, and the data cache. To ensure fair access to each of these resources, we force some processes to sleep for a short time: by

4. *ttcp* times the transmission and reception of data between two systems using the UDP or TCP protocols.

this mechanism, we guarantee that the operating system will perform a context switch. When any Disk Server process accesses a list or the cache, it first obtains a semaphore to guarantee exclusive access for the duration of the time it needs to perform its task. If other processes attempt to access the data, they are rejected and must, after a sleep-induced wait, try again.

Instead of the expensive semaphore mechanism, multiple threads guarantee exclusive access by using mutual-exclusion locks. The overhead of mutex locks is much less than that of semaphores, and checking mutex locks is much faster. Threads which cannot obtain a needed resource enter into a state of conditional waiting: this state eliminates the cycle of checking for the available resources, sleeping, and checking again, which characterizes processes attempting to gain a shared resource. Threads in conditional wait are simply put to sleep and signaled when the resource is available. Interthread communication is much faster than interprocess communication and threads consume fewer resources, since threads share the same text space with one another.

### 7.2 Real-Time Scheduling

An application like the Image Server System could benefit from real time scheduling. The ISS currently must attempt to coerce the UNIX scheduler to context-switch between the various competing ISS processes: trying to promote such context switching wastes time and reduces efficiency. A real-time operating system allows fine-grained control of the scheduler by means of thread prioritization and conditional waiting. Effectively, threads can take more or less processor time as necessary instead of arbitrarily taking a fixed slice of CPU time, and reducing competition with kernel-level or other user-level threads. This ability to vary the amount of time used by each thread is especially useful given that the ISS is driven by external events (the requests for the images) and must deliver the images back to the driving application within a predetermined time.

## 8.0 Experience

### 8.1 ATM Networking Issues

The design of the ISS is based upon the ability to use ATM network switches to aggregate cells from multiple physical data streams into a single high-bandwidth stream to the application. Figure 1 (Data Streams Aggregated by ATM Switches) shows multiple ISS servers being used to form a single high-speed data stream to the application.

Below is a list of what we have learned from our experience using ATM networks. Most of the experience reflected here comes from our work in the MAGIC gigabit testbed.

I) Hardware and Physical Layer:

- delivery of most ATM hardware has been delayed;
- the "infant mortality" rate has been high (several ATM interfaces and the ATM switch died in the first 60 days);
- it now seems clear that the workstation vendors have adopted multimode OC-3 as their preferred physical layer (we bought several 140 Mb/s TAXI, which were available six months ago);
- several of the multimode interfaces will drive single mode equipment (e.g., SONET terminals) by carrying the multimode fiber to the single mode interface (all examples of this have had active elements immediately behind the single mode interface);

II) Link Layer:

- not all of the ATM cell definitions (especially with respect to the Quality of Service (QOS) field) are uniform among manufacturers (QOS = 0 seems to be the point of agreement);

- there are many places where cell loss is apparent: these include switch output ports (next generation switches have much more buffering), and workstation interfaces, which are easily overrun for reasons not yet clear (it could be failure of kernel code to empty buffers fast enough);
- ATM device drivers are still fairly crude and buggy. This is especially true on multiprocessor systems, where the device drivers don't yet fully take advantage of the multi-threading capabilities of the operating system;

III) Network:

- Except for homogeneous switch environments, switched virtual circuits are not yet standardized, and in a large scale environment, use of PVCs makes set up and reconfiguration tedious and prone to errors;

IV) Transport:

- There are many throughput anomalies that are being investigated;
- Our work with timer-driven RTP shows some promise of it being a little more immune to cell loss than TCP.

One of the things that is becoming apparent in our work with this architecture is that the conventional notion of QOS is not a good method for regulating tightly coupled applications like the ISS, and (for similar reasons) may not be good for distributed-parallel compute server systems. Problems frequently occur when several servers that normally operate asynchronously to provide data to a single source, suddenly synchronize to produce a burst of data that overloads the switch and interface on the single receiver, thereby causing everybody to slow down and retransmit, leading to severe throughput degradation. This is very similar to the problem of routing message synchronization described by Floyd and Jacobson[2].

## 8.2 Memory Copy Speed

The main bottleneck for the ISS Disk Server is the speed of moving data into and out of memory. A SPARCStation 10, for example, has memory copy speed of about 22 Mbytes/s (176 Mbits/s). When writing to the network, the situation is even worse because data is moved to the interface via UNIX "mbufs" [7], adding additional overhead. We have measured the speed of an mbuf copy as 19 Mbytes/s (152 Mbits/s), and there are two mbuf copies required to send a packet to the network. Along with the other overhead required to assemble packets, this limits the speed with which we can write to the network to 9.2 Mbytes/s (74 Mbits/s).

If the network sends were faster, e.g., 19.4 Mbytes/s (155 Mbits/s - the OC-3 rate, ignoring ATM overhead), the next bottleneck would be the disk reading speed, which in this configuration is 9 Mbytes/s (72 Mbits/s). This bottleneck is trivially removed by adding more disks. This brings us back to the "memcpy" limit of 22 Mbytes/s as the key bottleneck. The other bottlenecks are not likely to be relevant in the near future. Increasing the speed of workstation memory is the key to increased performance for this application.

## 9.0 Conclusions

The emergence of wide area high-speed networks enables many types of new systems, include distributed parallel data servers. We have designed and implemented a special purpose high-speed data server, called the ISS. The ISS is designed to be distributed across multiple hosts with multiple disks on each host, and should be capable of scaling to gigabit per second rates. Moreover, we believe that the core design is flexible enough so that only minor modifications need be made to adapt the ISS to different data types and access patterns.

In the process of implementing and using this system, we have learned many things about workstation and operating system bottlenecks, and using ATM networks. The main things we discovered are:

- memory to memory copy speed is the main I/O bottleneck on today's workstations;
- ATM networks still have many problems to be worked out before they are ready for general use.

## 10.0 References

[1] Chen, L. T. and Rotem, D., "Declustering Objects for Visualization", Proc. of the 19th VLDB (Very Large Database) Conference, 1993.

[2] Floyd, S. and Jacobson, V., "The Synchronization of Periodic Routing Messages", Proceedings of SIGCOMM '93, 1993.

[3] Ghandeharizadeh, S. and Ramos, L, "Continuous Retrieval of Multimedia Data Using Parallelism, IEEE Transactions on Knowledge and Data Engineering, Vol 5, No 4, August 1993.

[4] Hartman, J. H. and Ousterhout, J. K., "Zebra: A Striped Network File System", Proceedings of the USENIX Workshop on File Systems, May 1992.

[5] Leclerc, Y.G. and Lau, S.Q., Jr.,"TerraVision: A Terrain Visualization System", SRI International, Technical Note #540, Menlo Park, CA, 1994.

[6] Langberg, M., "Silicon Graphics Lands Cable Deal with Time Warner Inc.", San Jose Mercury News, June 8, 1993.

[7] Leffler, S.J., McKusick, M.K., Karels, M. J. and Quarterman, J.S., "The Design and Implementation of the 4.3BSD UNIX Operating System", Addison-Wesley, Reading, Mass., 1989.

[8] Mills, D., "Network Time Protocol (Version 3) specification, implementation and analysis", RFC 1305, University of Delaware, March 1992.

[9] Mills, D., "Simple Network Time Protocol (SNTP)", RFC 1361, University of Delaware, August 1992.

[10] Patterson, D., Gibson, G., and Katz, R., "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in Proc. 1988 SIGMOD Conf., June 1988.

[11] Richer, I. and Fuller, B.B., "An Overview of the MAGIC Project," M93B0000173, The MITRE Corp., Bedford, MA, 1 Dec. 1993.

[12] Rowe, L. and Smith, B.C., "A Continuous Media Player", Proc. 3rd International Workshop on Network and Operating System Support for Digital Audio and Video, San Diego, CA, Nov. 1992.

[13] Schulzrinne, H. and Casner, S., "RTP: A Real-Time Transport Protocol", Internet Draft, Audio/Video Transport Working Group of the IETF, 1993.

# A 1.2 GBit/sec, 1 microsecond latency ATM interface

*Ron Minnich, Dan Burns, Frank Hady*

*Supercomputing Research Center*

*17100 Science Drive, Bowie, MD*

## Abstract

We are designing a zero-copy ATM interface call MINI, the Memory Integrated Network Interface. MINI has a target bandwidth of 1.2 Gbits/sec., or OC-24, and will support application-to-application latency of 1.3 microseconds. MINI is a multi-user ATM interface which resides in the memory space of the computer system, as opposed to I/O space. Applications will have direct access to pages of memory which are used for the network I/O; thus applications can send and receive packets with no operating system support needed. An application can initiate packet transmission with one memory write; it can determine if a packet has arrived by checking a per-circuit control and status word. At the same time, the operating system can use MINI for its own communications and for supporting standard networking software such as TCP/IP and NFS.

Initial structural VHDL simulations have shown that for single ATM cells we can 'bounce' an ATM cell from application to application in a round-trip-time of 3.9 microseconds, at a bandwidth of 10 Mbytes/second. For 960-byte packets, the bandwidth is 103 Mbytes/second.

The architecture of MINI is motivated by our experiences with applications we have run in cluster and distributed computing environments. We have used from 48 to 200 computers per application, and have observed that successful applications are ones that don't use the network much if at all. The MINI design is targeted at the kinds of applications we can not currently run, those that need to send or receive data at very frequent intervals.

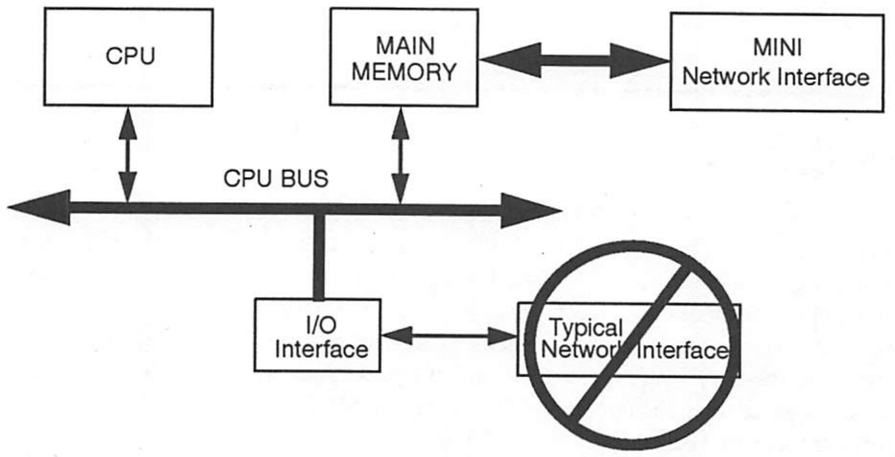
## Introduction

The limited bandwidth and high latency of current networks limits the set of cluster computing applications to those that require little use of the network. In our experience, to run a distributed computation on a network of 100 or more machines for five minutes only a few seconds of communications can be allowed. Thus the applications which run in a distributed computing environment must require only a byte of communication for every thousand or ten thousand FLOPS. Studies of typical applications by others[7] place the required application-to-application latency at about 1 microsecond. Additionally, our experience with multicomputer applications, of the type we would like to run in a cluster environment, shows that a large number of these applications require high network throughput.

New high bandwidth networks carry the promise of widening the scope of cluster computing applications to include those requiring substantial communications. Asynchronous Transfer Mode (ATM) in particular provides a clear long-term path to gigabit/sec bandwidths. Our experience with commercial network interfaces designed for these new networks has shown that their architecture is not suitable for a large set of distributed and cluster computing applications. These interfaces require operating system interaction each time a message is sent or received, greatly increasing latency and decreasing throughput. Our applications require an interface requiring no operating system involvement in sending or receiving messages while still allowing support for optimized versions of TCP/IP and NFS.

The network interface for the Memory Integrated Network Interface (MINI) architecture will be integrated into the memory hierarchy, not the I/O hierarchy, as shown in Figure 1. Typical implementations of network interfaces use the I/O bus. These interfaces suffer from bandwidth and latency limitations imposed by DMA start-up times, low I/O cycle times (compared to the CPU bus), limited I/O address space, multiple copying of data, and CPU bus contention. As we will show, these problems can be avoided by placing the network interface on the other side of main memory.

**FIGURE 1** MINI NETWORK INTERFACE



The ATM interface we are designing is being specified to meet certain performance goals. These goals are application-driven: that is, the architecture of the interface is being defined based on our experiences with applications that have succeeded (and failed) in a cluster and a distributed programming environment. The performance shown below is specified with respect to two workstations connected point-to-point. We do not consider a switched system in the goals as the ATM switch market remains immature.

The goals we have set for our interface are as follows:

1. Application-to-application latency on the order of 1 microsecond for small, single ATM cell, messages. (Latency is measured from the initiation of a host send command until the data is loaded into the target node's memory excluding fiber delay.)
2. Sustained application-to-application bandwidth of 1.2 Gbits/second for large (~4 kbyte) messages.
3. A Multiuser environment. (Measurements on 1 and 2 will not occur in single user mode)
4. Interoperability with an existing network standard (ATM).
5. Support for high performance (zero copy) TCP/IP and NFS.
6. Direct network access to 32 or 64MBytes of main memory.
7. Non-blocking polling of message receipt status.
8. Support for many Virtual Channels (4K VCs when using 4KB pages, 2K VCs for 8KB pages and 1K VCs for 16KB pages).

## Related Work

The architecture presented here originates from earlier experimental work conducted at SRC and at the University of Delaware. The idea of embedding the network interface in the memory architecture of the machine has its origins in the MemNet project[4]. MemNet was in essence a hardware distributed shared memory. A proposed addition to the MemNet architecture was the addition of a special region of memory which when written to could cause the movement of a selected piece of data from a source node to a destination node at the direction of an application[8]. Both the proposed modes were realized in software in the Mether Distributed Shared Memory[9][10][11], and more recently in Mether-NFS[15].

Hardware implementation of these ideas were considered in the implementation study of a network called SIMMNet[12][14] which embedded the data-driven semantics of Mether in hardware. The SIMMNet interface was designed for the SIMM slots of SPARCStations. The target bandwidth, application-to-application, was 800 Mbits/second, or memory bandwidth. Many ideas found in SIMMNet have been applied to the current design.

A variety of high bandwidth network interfaces appearing in the literature are described below. The emphasis in these interfaces is primarily on achieving high bandwidth. These interfaces are designed to function in a Wide Area Network, so latency is not a primary consideration.

In Van Jacobson's Witless[6] interface careful attention is paid to page-aligning data as well as reducing the number of copies. In the spirit of Witless, a group at HP Labs[2] created a single copy interface called Afterburner. Here, 1 Megabyte of VRAM mapped to I/O space houses the kernel buffers user for communication. TCP/IP performance of 210Mbits/sec was demonstrated.

Smith and Traw[17][19] have demonstrated that careful interface design, coupled with attention to OS details, can lead to high performance. Their interface, demonstrated at 155 Mbits/sec, consists of a segmenter capable of DMAing packets from memory, and a reassembler capable of constructing packets from ATM cells. Clocked interrupts are used to avoid unacceptable interrupt cost. Smith used 'pinned buffers' (i.e. non-pageable virtual memory) directly accessible from both the interface and the application, to reduce the number of copies.

Davie[3] presents an interface positioned on the DEC Turbochannel I/O bus featuring two i960s, one for segmentation and the other for reassembly. This interface is targeted at a throughput of OC-12 (622 Mb/sec), but is limited substantially by the overhead in DMAing 48 byte cells.

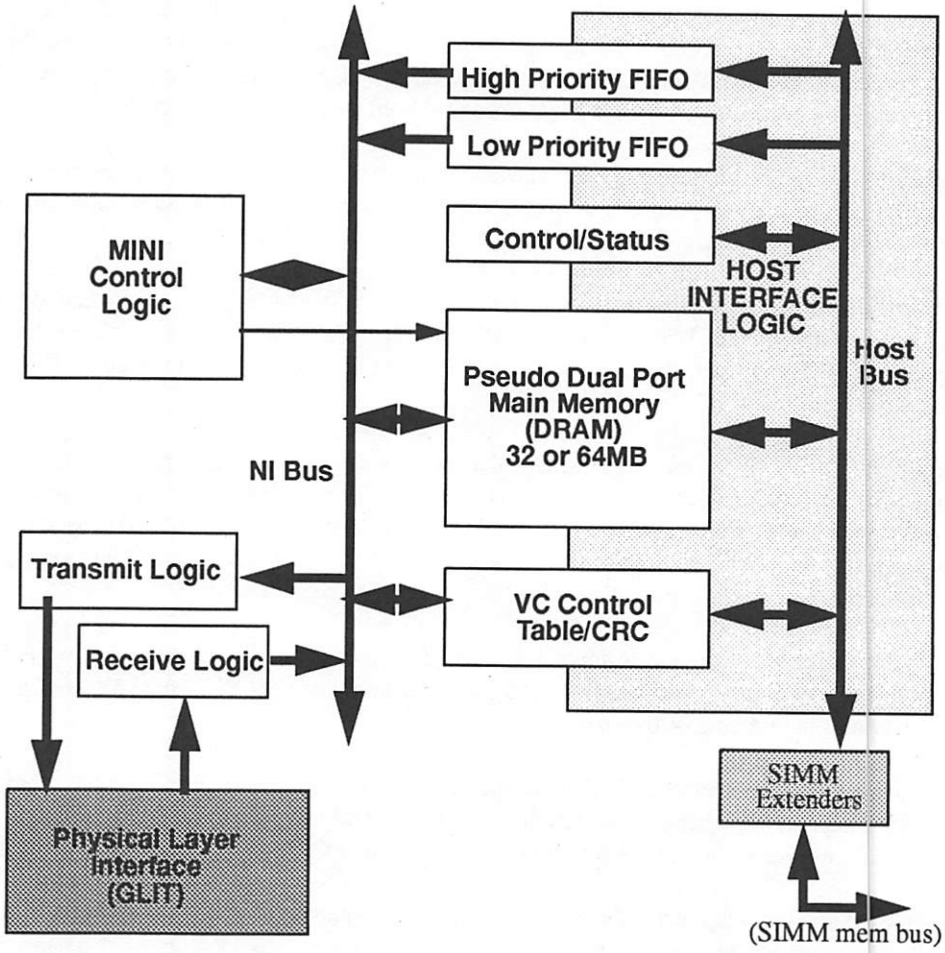
Larry Peterson has achieved 255 Mbits/sec. performance using pinned buffers ("fbufs"), showing that even on a complex interface high performance is possible[20]. Tennenhaus et. al. have demonstrated a "network co-processor"[18], that supports real-time video. This processor is connected to the host CPU via the cache-line interface.

Unlike the interfaces described above, the Shrimp Multicomputer[1] also considers latency, targeting a single word latency of 2 microseconds. This interface has a virtual memory mapped network interface that supports communication at the user level in order to avoid operating system interaction. Shrimp uses a network interface page table with an entry for each page of physical memory. This table specifies the destination node and page number to which each local page is mapped. In this interface virtual memory remapping is a global problem, that is a destination page can not be re-mapped to another location in memory without knowledge of the sending process. For its deliberate update mode, Shrimp uses DMA to increase bandwidth to 264 Mbits/sec. Shrimp has been designed to connect with a multicomputer network, an Intel Paragon routing backplane.

## Hardware Architecture

The major components of the MINI architecture are shown in Figure 2. The architecture has been kept as simple and low cost as possible without sacrificing target latency, target throughput and multi-user access. MINI has several memory-addressable areas that allow communication between the host and network interface. The shaded block labeled Host Interface Logic controls host access to these memory-addressable areas. This block is the only part of the architecture that must be redesigned when porting the design to another host.

FIGURE 2 MINI Architecture



The interface allows for many individual channels, corresponding to ATM Virtual Channels (VCs), into the network. A process (user or OS) may use one or more channels. The VC Control Table contains two entries for each VC, a send control word and a receive control word. Each control word is loaded with a pointer to an area in Main Memory (a double page located wherever the kernel and process agree) as well as status, ATM cell header, and CRC information. Control words are written by the operating system whenever a process requests a channel into the network. Also as part of the channel setup process, the double page pointed to by the control word is allocated to the requesting process. A maximum of 4K VCs will be supported when page size is 4KBytes with 2K VCs for 8KByte pages and 1K VCs for 16KByte pages.

After channel setup is complete, the sending process (user or OS) initiates a send with a single write to either the High or Low Priority FIFO. The High Priority FIFO is used for messages that must retain low latency in the presence of large amounts of network traffic. The word held by the High or Low Priority FIFO contains a VC number, a start address within the channel's double page, and a stop address within the channel's double page. Using the start and stop addresses any amount of data and/or protocol information up to the size of the double page can be sent. This scheme allows a page of data sent with a header and trailer (i.e. a page send using TCP/IP or NFS) to arrive page aligned, negating the need for a copy on the receiving node. The NI Control Logic uses the VC number from the High or Low Priority FIFO as an index into the VC Control Table. Channel information, such as the double page address and ATM header field values, is retrieved from the VC Control Table. Next, the ATM header is constructed and sent, a Main Memory address is constructed, data from Main Memory is retrieved and written to the Transmit FIFO 64 bits at a time. Finally, the VC Control Table send word is updated. The interface continues to send cells into the network on this VC whenever possible until the entire message has been sent.

When a cell arrives from the network, the header is stripped and the cell's data is placed in Main Memory at the location indicated by the VC Control Table Entry for the VC on which the cell arrived. The VC Control Table entry is then updated to prepare for the arrival of the next cell. Arrival status can be checked by the receiving process (user or OS) at any time.

### Performance

ATM cells are packaged and transmitted at a rate of 1 cell every 400nS, or 120MB/s (0.96Gbits/s). (Note that this is the real data transfer rate. The total bit transfer rate, including ATM header information is 140MBytes/s (1.12Gbits/s).) The G-Link Translation board (GLIT) removes the cells 64-bits at a time from a transmit FIFO at a rate of 150MBytes/s and sends them 16-bits at a time (at 75MHz) to a Finisar Transmit Module which serializes and encodes the data, adding 4 bits to every 16, and optically sends the data out onto the fiber link at a rate of 1.5GBaud.

### Flow Control

In hardware MINI will support a low level flow control designed solely to keep the network from dropping many cells. In this scheme a STOP message is sent from a receiving node to the transmitting node (or switch port) when the Receive Logic FIFO of the receiving node goes past some pre-determined high water mark. Transmission will be restarted by sending a START message to the transmitting node (or switch port) when the same FIFO falls below a pre-determined low water mark[16].

Support for software implemented flow control is also include in the MINI architecture. In the event that it becomes necessary to drop incoming cells, the MINI hardware keeps a count of the number of cells dropped on a per VC basis within the VCI Table. Real Time Clock values automatically passed within messages may also prove useful.

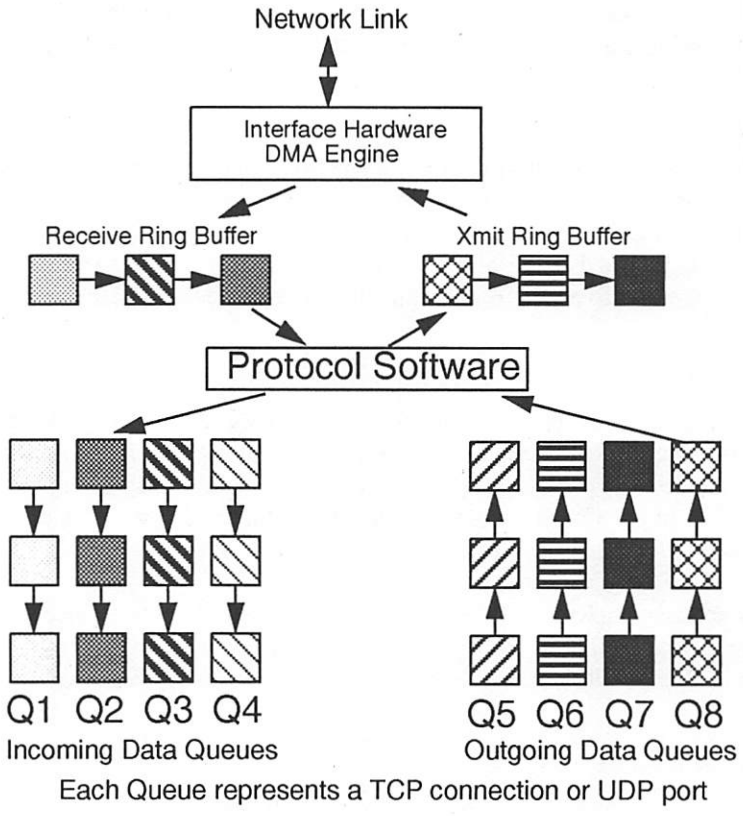
## Software Architecture

Before discussing the MINI software, we will quickly recap some key differences between MINI and other network interfaces. These differences have an impact on how the software is structured.

MINI was designed with a number of goals in mind, one of the most important being sustained high bandwidth transfers. Some of the most difficult issues in this area involve buffer pool management, interrupt handling, and device I/O, especially DMA devices. All of these activities are time-consuming: our measurements on several architectures show that the cumulative overhead of these activities is several hundred microseconds per Ethernet packet at a minimum; no workstation we have measured has come even close to 250 microseconds. At the data rates we plan to use, one 4K packet can come in every 30 microseconds. Fielding an interrupt for each of these packets is impractical. The data rate does not even allow much overhead per packet, even if we only allow an interrupt every 10 packets or so.

Most network interfaces today use the model of a set of buffers, managed in a ring, which are either emptied or filled by the network interface. As the buffers are filled or emptied, the interface interrupts the processor to signal buffer availability. The processor can then use the available buffers as needed, reusing them for output or taking them as input and attaching them to other queues. An example system is shown in Figure 3. The protocol software is managing queues of incoming and outgoing data. Each queue represents either a UDP port or a TCP socket. As ring buffer slots become available, data is copied or in some cases "loaned" to or from the ring buffers so that the DMA engine can access it.

**FIGURE 3** A typical network interface structure, with incoming and outgoing queues and hardware ring buffers.



The management and control of these buffers is time-consuming, as is the necessity of handling the associated interrupts. In these interfaces, the DMA engine is a single-threaded resource, which means the system software must manage the time-sharing of that resource, adding further overhead. Mapping virtual to physical addresses can be a large source of overhead. Virtual address-based DMA systems actually tend to be more expensive due to the necessity of setting up I/O mapper hardware.

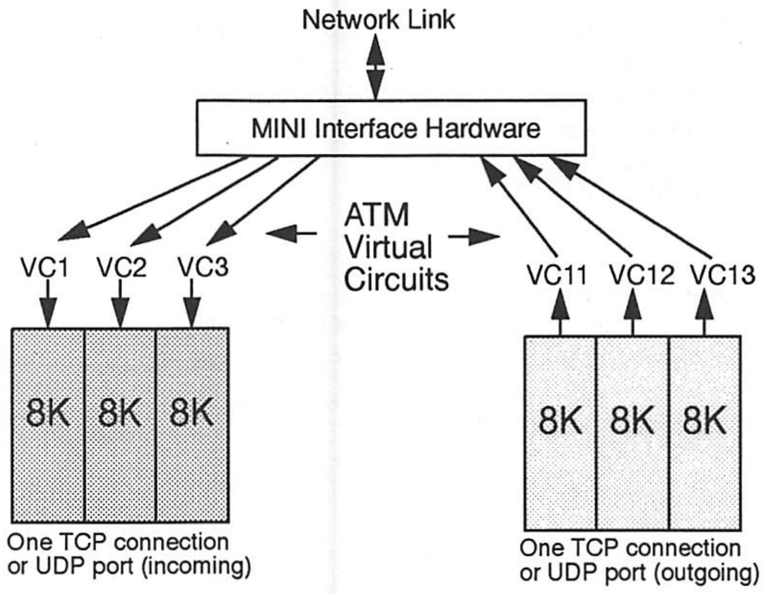
All these overheads add up, sometimes to a surprisingly large number. For example, on an SBUS interface we built at SRC we found that only 20 Mbytes/second of the 48 Mbytes/second of SBUS bandwidth could be obtained; the other 28 Mbytes/second was consumed in DMA setup and OS overhead.

On MINI we are determined to reduce the overhead by reducing the complexity of the data path from the network to the application. In the best case processes on different machines should be only a few memory writes away from each other in time, and this low latency is hard to accomplish if there are queues and linked lists between them.

As mentioned above, MINI eliminates many of the overheads associated with buffer pools by not having buffer pools. The interface supports up to 4096 DMA channels, one per virtual circuit, but each channel can only go to an 8K area. Part of the process of setting up a virtual circuit involves allocating the area of memory associated with that circuit. What then happens to the buffer pools? Our intent is to, in effect, turn the buffer pools sideways. That is, for applications or protocols that need more than 8K of receive or send buffering we expect the application to allocate multiple ATM channels. Thus, instead of looking down a long pipe at a stream of buffers coming in from an interface, one can image looking at an array of buffers standing shoulder-to-shoulder, as shown in Figure 4.

Removing the linked list management and other hardware from the network interface has a nice side-effect: the interface becomes very simple. The ring buffer is a complex structure to deal with in hardware, made more difficult by the fact that both the host and the network interface need to access the structures, and thus require memory-based interlocks.

**FIGURE 4** How MINI uses multiple ATM circuits to support buffering on a connection or port



Another feature of MINI is designed with existing protocols in mind. The software on the receiving end can set the starting offset for the data in the 8K buffer. Once the PDU data reaches the end of the buffer, the counter wraps so that the tail of the PDU is written to the beginning of the buffer.

MINI works this way so that we can support a zero-copy NFS. We plan to have a modified version of NFS that has a fixed-size header, fixed page (4096 bytes) of data, and variable length trailer. By our measurements fixed-size page-size data packets account for 66% of the data moved on our networks, so efficiently supporting these data packets is crucial.

NFS clients can arrange their requests so that the returned page data lands on a page-aligned boundary by setting the offset of the receiving channel. Thus, once a page is received from a server, it never moves again. This technique will decrease the overhead for an NFS read. We can also employ the same technique on the server side: we can arrange to have the data read in from the disk to the second page of the 8K block, and build the NFS header and trailer in the first page. In fact, it should be possible to perform an NFS read in less than 150 microseconds, for pages on the server that are frequently accessed and are hence in memory. This contrasts with the 4.5 milliseconds it currently takes on Ethernet-based servers.

Of course, this zero-copy bulk data transfer technique can also be used for TCP clients. They could, for example, have an array that was virtually contiguous (so it looked like a Fortran block common, for example), but physically non-contiguous. PVM or MPI could use this property to initiate transfers of the array in whole or part to other machines.

We will now go into some of the low-level details of interest to C programmers or those building C++ objects for using this interface, following which we will explore the protocol issues in more detail.

## Low-level details for C programmers

The interface presents a uniform appearance to both kernel mode and user mode software. For an active virtual circuit, there are two control words, one for incoming and one for outgoing data. There is also at lease one pair of pages, aligned on an 8K boundary, for the virtual circuit. There can be two pairs, one for outgoing data and one for incoming data. Thus, each link can be half or full duplex. C structures corresponding to the control words are shown in Figure 5 and Figure 6.

The packing of the bits into one 64-bit word may seem unnecessarily constrained. We have packed them this way out of necessity. There is a subtle software issue with respect to our emphasis on making the bits fit in one word. Some of the accesses to this data will be from user mode. There is no guarantee given to a user program that it can load two consecutive words of memory in two consecutive instruction times. A program could load one word, be interrupted for an indeterminate time, and then when it is next scheduled load the next word. The two words could be out of sync with respect to each other. To avoid the potential problems caused by such interrupts, we have made sure that all the bits for a given control word can fit in one word. We would like to have more bits in some cases, but have determined that we can manage with what we have.

**FIGURE 5** C structure for receive control word

```
struct receiveword
    [
     long long
        spare: 3,
        pducount: 4,
        status_inuse: 1,
        status_headererror: 1,
        status_inactivevciused: 1,
        status_packarrivingcells: 1,
        /* number of cells dropped on this VCI, 10 bits,
         * bit 10 is sticky
         */
        droppedcellcount: 10,
        /* double-page(8k) pointer, PFN, 15 bits */
        pagepointer: 15,
        /* Current NI index into double-page, 12 bits */
        wordcount: 12,
        aal5middlehec: 8,
        aal5endhec: 8;
    ];
```

The bits most commonly used are those corresponding to the pducount. In user mode, the application can simply repeatedly sample these bits, waiting for them to change. Such polling will have no impact on the network bandwidth of the MINI interface, because the polling and data transfer are done on two separate buses. The timing of the MINI memory is such that for every word or cache line read performed by the host one ATM packet can be received or sent. Polling can have an impact on the host, however; we discuss strategies for dealing with this situation below.

The pducount is mainly intended for use for large PDUs, given the small number of bits we provide. These larger pdus will fill most of the 8K space, and so one PDU corresponds to one 8K doublepage. The intent is that a sending side will never send a large PDU until the receiving side has given permission (or in TCP jargon, 'opened the window'). Thus the pducount can to a limited extent allow software to determine that there has been an error in the software, and we expect a TCP sequence number or equivalent to indicate just how much data was lost.

For small PDUs, in a non-windowing protocol, the word count can be used to determine how much data has come in and if there has been an overflow. There are enough bits in the word count to count 1 Mbyte; this is 8 milliseconds at our 1.2 Gbit/second rate, which implies that we need to be able to schedule the receiving process at this interval. Eight ms. is less than the Unix scheduling quanta but on a system with real-time scheduling extensions such as the Indy there should be no problem.

The circuit can be polled for incoming data, but for large numbers of circuits this could be very inefficient. The interface provides a select mask of 64 bits, where each bits represents 128 ATM circuits. The operating system will need to allocate circuit numbers in such a way that not all the active circuits correspond to one bit in the select mask.

**FIGURE 6** C structure for send control word

```
struct sendword
    {
        long long
            spare:2,
            /* upper 4 bits of vci number */
            upper_vci:4,
            /*ATM Generic flow control,4 bits*/
            GFC:4,
            size:10,
            /*double page pointer, PFN */
            pagepointer:15,
            /*Current NI index into doublepage,12bits*/
            wordcount:12,
            aal5middlehec:8,
            aal5endhec:8;
        };
```

The fields most commonly used in the send control word are the size, and wordcount fields; in kernel mode, the page_pointer field may change frequently or not at all, depending on the use of the channel.

Message transmission requests are written to the high or low priority FIFOs. Messages can range in size from one cell to 170 cells. The hardware will start at any point in the doublepage, as indicated in the request, and wrap around and continue sending from the start of the doublepage. The VCI, size, and offset are placed in a word as shown in Figure 7.

**FIGURE 7** Format of the Message Transmit Request Word.

```
struct sendrequestword
     {
          long long
              /* note: 36-bit fifos, so no hardware
               * for the unused bits!
               */
              unused: 28,
              /* vci is only set in kernel mode --
               * set by hardware in user mode
               */
              vci: 12,
              size: 12,
              offset: 12;
          };
struct sendrequest
     {
          struct sendrequestword sr_word;
          unsigned long long sr_long;
     };
```

The software structure which defines a channel is shown in Figure 8.

To send a message, in the worst case, the software needs to compute an offset in the 8K area in which to put the data; determine the size; copy in the data; set the offset and size parameters in a Send Request Word as shown in Figure 7; and write to the Send Fifo. The message will be sent, and the PDU counter in the Send Control Word will be incremented when the message is gone.

To send in what we hope will be the common case, the software simply composes the send request word and writes it to the high or low priority FIFO. No data copying is needed since the data is in place. The software may need to write a header and trailer, as in the case of NFS or MNFS.

**FIGURE 8** Structure for maintaining a virtual circuit

```
struct vc
     {
          /* send status word */
          struct sendword *send;
          /* receive status word */
          struct receiveword *receive;
          /* pointers for high and low priority
           * messages
           */
          unsigned long *lowp, *highp;
          /* (CURRENTLY NOT IMPLEMENTED)
           * select word for seeing which groups of
          * circuits have incoming data
          * bit <n> signifies circuits in the range
          * <n>*128 => <n+1>*128 - 1
          unsigned long long *select;
          */
          /* note that the recpage and
           * sendpage can point to
           * the same place, or different
           * places
           */
          caddr_t recpage, sendpage;
     };
```

## Virtual Multicast

Because the data associated with a VC can come from any page pair in memory, we can associate a single page with many VCs. We can fill out the data once, then send the data on each VC by simply writing to the high or low priority FIFO. Thus the cost of sending the same <N> byte message on <M> channels is <N> + <M> memory writes. We call this technique "virtual multicast", since it has some of the advantages of multicast and is fairly easy to use.

## Direct application Access to the Interface.

As shown in Figure 9 an application can directly access the hardware if necessary. The code to transmit data is simple, and multiple applications can use multiple channels simultaneously. The memory area is large enough that it could, in practice, comprise the entire memory of the machine.

Note that this is an example of an inefficient use of the interface, since the send data originates elsewhere. The value that is written to the FIFO pointer, highp, is unimportant, since the interface actually determines the circuit number from the address used. Using the address to encode the circuit number ensures that applications can only do sends on circuits they have been granted access to.

The application can wait for all the data to be sent, as shown in lines 10 and 11, by checking until the wordcount is greater than the size. This can take anywhere from 500 nanoseconds to 71 microseconds or more; applications that are sending large packets should be willing to allow other applications to run. In most cases we expect users to use TCP for bulk data transfer.

**FIGURE 9** Sample applications code for sending single-cell streams

```
1  int atmsend(vc, data, nbytes)
2  struct vc *vc;
3  caddr_t data;
4  unsigned long nbytes;
5  {
6      struct sendrequest sr;
7      sr.sr_word.size = nbytes;
8      sr.sr_word.offset = 0;
9      bcopy(data, vc->sendpage);
10     *vc->highp = sr.sr_long;
11     while (vc->send.wordcount < vc->send.size)
12         ;
13 }
```

## Barrier Synchronization

Many parallel applications at some point synchronize the component processes in order to trade intermediate results or stabilize the state of the application prior to moving to a new activity. This synchronization is typically accomplished by setting up shared state, and requiring every process to update that state before any one process can proceed. This state can be thought of as a wall, or barrier, at which all the participants must arrive and regroup before proceeding; the technique is typically called barrier synchronization.

Typically the participants in the computation will increment a counting semaphore or, in a message-passing environment, send a message to some other designated process. Once the designated process has received messages from all the participants, its sends out a message to them all allowing them to continue.

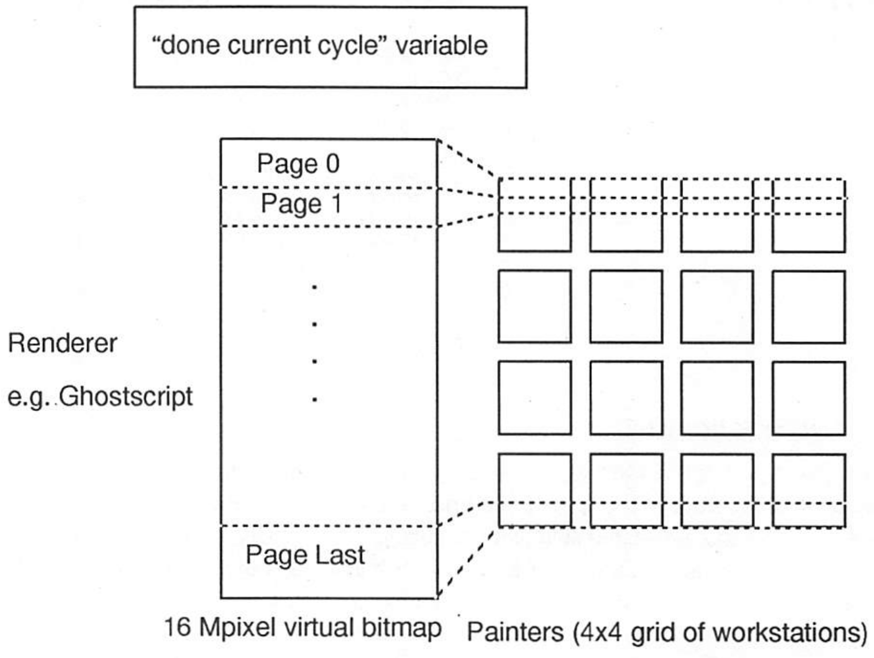
The single-cell messages are particularly useful for barrier synchronization. The individual programs would set up half-duplex links to the master program. When they were ready to synchronize they would send one cell on the link. The master can wait as the single-cell messages come in, until the total is reached. The cost in time per cell is 1.2 microseconds. The master can then use virtual multicast to send one "go" cell to all the other programs, at a cost of 1.2 microseconds per program. Thus we can perform a barrier synchronization using this simple algorithm at a cost of 2.4 microseconds per participant.

For barrier synchronizations involving more participants we can arrange them in a tree, and lower the cost. In any event these costs are comparable to those found on an MPP such as a CM-5.

## Network Graphics Example

The example send code shown above does not take full advantage of the capabilities of the interface. In particular, there is no use of the capabilities of the interface for supporting transfer of bulk data to many different processors. We will illustrate these capabilities with an example of network graphics, based on the work described in [15], in which we used an array of workstations, assembled into a custom rack, to display pictures as though they were one large bitmap. We used the MNFS distributed shared memory to support the large virtual bitmap, and on each workstation ran a program which did a bit-block transfer from the large virtual bitmap to the workstation's frame buffer. The layout of the renderer (which draws into the 16 Mbyte picture bitmap) and the painters (which copy from the picture bitmap to the framebuffers) is shown in Figure 10. The renderer renders into the 16 Mpixel bit map, and using the msync system call flushes all modified pages at the end of a cycle, then increments the "done current cycle" variable when it is completed. The painters wake up when the "done current cycle" variable changes, and render into the frame buffers.

**FIGURE 10** Layout of the renderer and the painters in distributed shared memory.

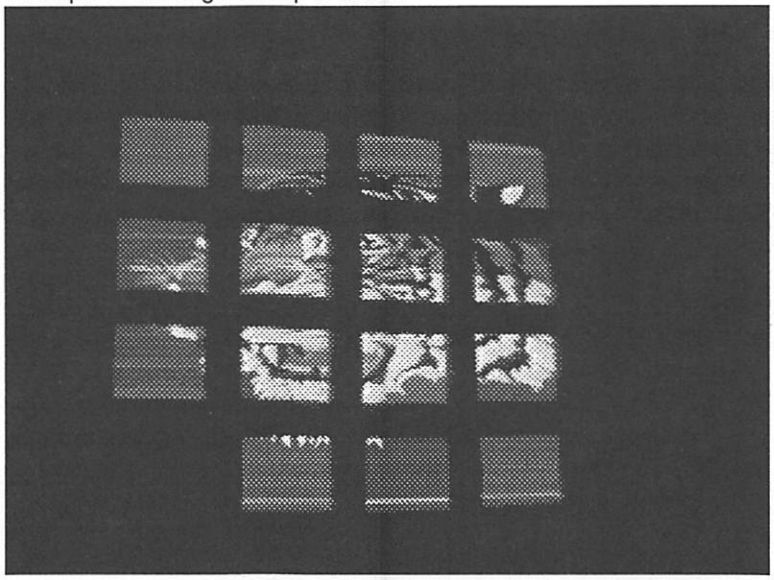


An example of the system in action is shown in Figure 11. The figure is the tiger image that comes with the ghostscript distribution. The picture was taken in a darkened room using an IndyCam.

This program could change to some extent in the ATM version[1]. One immediate candidate for change is to replace the "done current cycle" variable by a direct ATM connection using the code shown above. We could for example use the barrier synchronization described above. MNFS would then be used only for the actual bitmap pages.

A higher-performance system, at higher cost in the number of connections, could also be employed. When we move to the color version of this display the bitmap will require 16 Mbytes. This will require 4096 pages of memory. As shown in the figure, each page is shared across four machines. Thus each machine needs part of 1024 pages for its bitmap. We could open up 128 connections to each machine, and send the 1024 pages in groups of 128. In the worst case, i.e. clear the bitmap and repaint every pixel, 16384 pages will need to be sent to the various hosts. Each 4096 byte page takes 40 microseconds, so the total time to transfer the entire 8-bit bitmap will be 0.66 seconds. In practice the entire bitmap is not being dirtied every time; we can achieve a frame rate of 24 frames per second if only a megabyte or so of the virtual bitmap is dirtied each frame. At the receiving end the pages can be placed such that they are virtually contiguous, even if not physically contiguous, and the block transfer can be a simple bcopy. Note that we can also add more MINI interfaces to the computer if a higher frame rate is desired.

**FIGURE 11** Example of the large bitmap in action



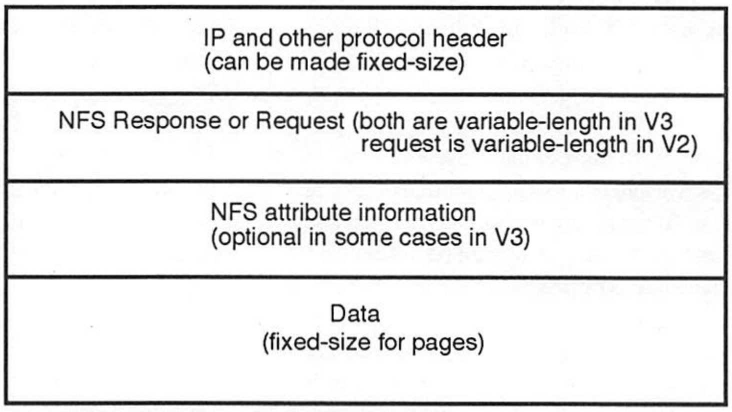
## Zero-copy NFS

We plan to implement a zero-copy NFS using this interface. In a zero-copy NFS, data that is transferred between client and server lands in memory page-aligned, so that no copying is needed to make the data accessible to the applications or to other I/O subsystems. The MINI interface has been designed with this type of NFS modification in mind.

Shown in Figure 12 are typical packets for NFS versions two and three. The data in most cases follows variable-length packet header information. This variable-length header is not a problem in current networks and NFS implementations due to the minimum of one copy that must be made from the incoming NFS packet to the final resting place of the data. In part this copying is necessitated by the fact that most network interfaces can not ship a full-sized NFS packet as a unit, with the result that a zero-copy implementation has never been practical.

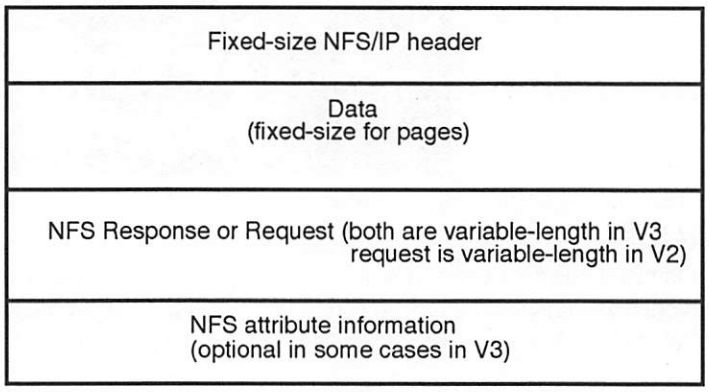
1. The original MNFS version could be used as is, of course.

**FIGURE 12** A Typical NFS read response or write request packet.


IP and other protocol header
(can be made fixed-size)
NFS Response or Request (both are variable-length in V3 request is variable-length in V2)
NFS attribute information
(optional in some cases in V3)
Data
(fixed-size for pages)

Shown in Figure 13 is the modified NFS packet with the fixed-size data placed in the middle of the packet. Because MINI can work with packets that start in the middle of the buffer and wrap around to the front, we place the packet so that the data is page-aligned, as shown in Figure 14. When the packet is sent over the fiber, the fixed-size NFS header is sent first, followed by the data, followed by the variable-sized trailer. Thus the packet moves over the network header-first, but once it arrives at MINI it is stored to memory in such a way that the trailer comes first in memory, followed by the header, followed by the data.

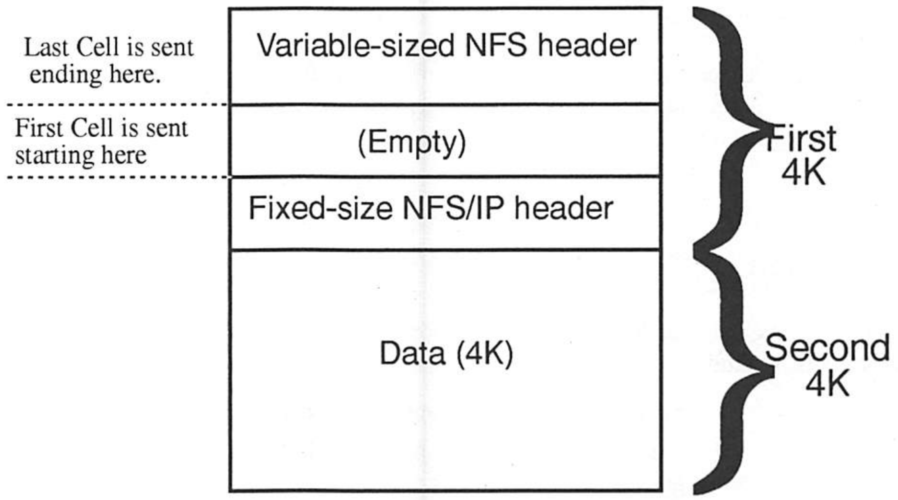
**FIGURE 13** Modified NFS packet with fixed-size data in the middle and arranged in the two-page buffer.


Fixed-size NFS/IP header
Data
(fixed-size for pages)
NFS Response or Request (both are variable-length in V3 request is variable-length in V2)
NFS attribute information
(optional in some cases in V3)

The data-first arrangement of a packet has been tried before, in a format known as "trailers". The advantages were the same: the data, sized at 512 bytes, landed on page-aligned boundaries on the machines in question, which were VAXes with 512-byte pages. There were many problems with trailers, not the least of which being that most machines which followed had larger pages which would not fit in one Ethernet packet. Thus, the use and benefits of trailers were short lived.

While our arrangement has the benefits of trailers, in that the data lands page-aligned, it has an important advantage over the trailer technique of simply sending the data first followed by the header,: the data can be sent to or from non-MINI interfaces, and MINI will still be able to use the optimized storage format and function as a zero-copy interface.

**FIGURE 14** Placement of the modified NFS packet in MINI memory.


Last Cell is sent ending here.
First Cell is sent starting here
Variable-sized NFS header
(Empty)
Fixed-size NFS/IP header
Data (4K)
First 4K
Second 4K

## Overlapped transfers

User programs can initialize the next ATM cell while the current one is being output from the interface. This allows overlapped transfers and would allow an application to source ATM packets at the rate of one per microsecond. For bulk data transfer, however, additional considerations come into play and we anticipate that users would use TCP for this purpose.

## TCP

We are working on an a zero-copy version of TCP for this interface. A zero-copy implementation of TCP should be able to fill the MINI network buffers at a rate of 40 Mbytes/second, i.e. half the memory bandwidth of the Indy. Because the bandwidth of MINI is 2.5 times this number, TCP will be able to use a double-buffer scheme by setting up two full-duplex connections, and filling the data area for one connection while the other connection is actively sending data. Thus two machines can communicate at full memory bandwidth over two full-duplex MINI ATM connections.

## Caching Issues

Because of the direct user-mode access to the interface, and because our I/O is not cache-coherent, some areas of the interface can not be cached. In particular, the send and receive control words can not be cached. In some cases, the data areas will be cached, and in others they will be used uncached.

## Status

A VHDL simulation of MINI has been working since May 1994. The simulator is structural down to the component level and includes realistic component models. The VHDL simulator is currently driven from C++ code[5], allowing simultaneous hardware and software development. Control state machines continue to be refined toward future implementation in programmable parts. Measurements have shown application to application latencies of 1.3 *u*sec for 1 cell messages and large message throughputs of 1 Gbit/sec for larger messages.

The physical layer specific card (called GLIT) is finished and will arrive by August. The hardware design for the version of MINI which will plug into a Silicon Graphics Indy began in May. Current plans are to have a two node system working by the end of the year. The Operating System drivers, and kernel modifications required for fast NFS and TCP/IP are underway.

## Conclusion

In this paper we have described MINI, a Memory Integrated Network Interface. MINI has been designed with a number of goals in mind:

1. Support application-to-application latency of 1.3 microseconds, point-to-point.
2. Support application-to-application bulk data transfer bandwidth of over 1 Gbit/second, i.e. greater than HIPPI rates.

Although MINI provides low-latency paths, they are not low bandwidth paths; conversely, high bandwidth paths exhibit fairly low latency. When applications are using the 1.3 microsecond path, they can still realize bandwidth of 200 Mbits/second; applications transferring data at 1 Gbits/second can see a latency as low as 12 microseconds.

MINI allows very low-cost barrier synchronization, useful for parallel programs. MINI also allows a "virtual multicast": applications which need to send <M> bytes of data to <N> hosts can do so at a cost of <M>+<N> memory writes, rather than the O(<M>*<N>) memory writes typical of most host interfaces.

Specific features of the MINI architecture have been designed with existing network protocols in mind, particularly those which allow fix-sized data in the packet to be page-aligned. Using these capabilities we are implementing a zero-copy NFS.

MINI simulation is being done in structural VHDL, and we have C++ programs driving the simulation. These same programs will drive the hardware: thus the programs used to validate the simulator also form the first set of programs we will run on the actual system.

As of June 17, 1994, some MINI cards have been sent out for fabrication and we are waiting for their return; the simulation is allowing programs to move packets; and the final MINI card design is well underway. Sections of the MINI architecture will become operational starting in August with a full point to point system running by the end of the year.

## References

1. Blumrich, Matthias, Kai Li, Richard Alpert, Cezary Dubnicki, Edward Felton, Johathan Sandberg, "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer", *The 21st International Symposium on Computer Architecture,* May 1994, pp. 142, 153.
2. Dalton, Chris, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, Lohn Lumley, "Afterburner'" *IEEE Network,* July 1993, pp.36,43.
3. Davie, Bruce S., "The Architecture and Implementation of a High-Speed Host Interface," *IEEE Journal on Selected Areas of Communication,* Vol 11 No. 2, February 1993 pp. 228,239.
4. Delp, Gary, "The architecture and implementation of memnet: A high-speed shared memory computer communications network", Ph. D. Thesis, University of Delaware, Dept. of Computer Science, 1988.
5. Fross, Brad, "Modeling Techniques Using VHDL/C - Language Interfacing", *VHDL International Users Forum Conference,* 1993.
6. Jakobson, Van, "Witless interface", presented at the 1991 Gigabit Workshop, Washington, D.C.
7. Kolawa, Adam, Bryan Strickland, "Automatic Parallelization of Programs for Distributed Memory Systems: A Case Study", *Cluster Workshop '93.*
8. Minnich, Ron, "Extensions to MemNet", Personal note to Gary Delp and Dave Farber, 1987.
9. Minnich, Ron, Dave Farber, "The Mether System: A Distributed Shared Memory for SunOS 4.0", Usenix, 1989.
10. Minnich, Ron, Dave Farber, "Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory", Proceedings of the Tenth IEEE Distributed Computing Systems Conference, 1990.
11. Minnich, Ron, "Mether: A Memory System for Network Multiprocessors", Ph. D. Thesis, University of Pennsylvania Department of Computer Science, 1990.
12. Minnich, Ron, Daniel Pryor, "Radiative Heat Transfer Simulation on a SPARCStation Farm", *Concurrency: Practice and Experience,* 1993
13. Minnich, Ronald, Dan Burns, Frank Hady, "An OC-24, 1 microsecond ATM Host Interface," Supercomputing Research Center TR-93-111, Oct 1993
14. Minnich, Ron, "SIMMNet", personal note to Larry McVoy of Sun Microsystems, 1992.
15. Minnich, R., "Mether-NFS: A Modified NFS which supports Virtual Shared Memory", *SEDMS 93.*
16. "Myrinet Link Specifications," Myricom Inc., December 10 1993. provided by Chuck Seitz.
17. Smith, Jonathan M. and C. Brendan S. Traw, "Giving Applications Access to Gb/s Networking", *IEEE Network Magazine,* Vol. 7, No. 4, July 1993, pp. 44-52
18. Tennenhaus, D. L., "ATM Co-processor", presented at the Aurora Jamboree, 1991.
19. Traw, C. Brendan S., Jonathan M Smith, "Hardware/Software Organization of a High-Performance ATM Host Interface," *IEEE Journal on Selected Areas of Communication,* Vol 11 No. 2, February 1993, pp, 240,253.
20. Peterson, Larry, "Is There Life After MicroKernels?", Keynote Speech, *SEDMS '93*

## BIOGRAPHIES

### Ron Minnich

Ron Minnich is a Research Staff Member at the Supercomputing Research Center. Current work includes MINI; MNFS, a modified NFS which supports distributed shared memory; and operating systems modifications to support high-performance distributed and cluster computing on from 32 to 500 machines.

### Frank Hady

Frank Hady is currently a Research Staff Member at the Supercomputing Research Center in Bowie, MD. At SRC he helped to design and build Hnet, the High-speed Network Evaluation Testbed. He has used Hnet to study interconnection network performance with traffic generated by real parallel applications. He has also developed analytical models to accurately predict the performance of a variety of wormhole routed networks. His current interests include MINI, the application of multicomputer interconnection network switching techniques to LAN switches and the creation of realistic network traffic models.

### Dan Burns

Mr. Burns has 11 years experience as a hardware designer. As a Research Staff Member at the Supercomputing Research Center he has worked on designs of MINI, HNET, a high-speed interconnect network for 64 processing nodes and SPLASH2 a reprogrammable, FPGA-based systolic array processor. Prior work at Northrop Corporation includes a T4/T3/T2 programmable demultiplexer for Time Division Multiplexing systems, a portable Doppler radar design and lead systems engineer of a $3M communication system.

# XTP as a Transport Protocol for Distributed Parallel Processing*

W. Timothy Strayer Michael J. Lewis Raymond E. Cline, Jr.

*Distributed Computing Department*
*Sandia National Laboratories, California*
*{strayer,mlewis,rec}@ca.sandia.gov*

**Abstract**

The Xpress Transfer Protocol (XTP) is a flexible transport layer protocol designed to provide efficient service without dictating the communication paradigm or the delivery characteristics that qualify the paradigm. XTP provides the tools to build communication services appropriate to the application. Current data delivery solutions for many popular cluster computing environments use TCP and UDP. We examine TCP, UDP, and XTP with respect to the communication characteristics typical of parallel applications. We perform measurements of end-to-end latency for several paradigms important to cluster computing. An implementation of XTP is shown to be comparable to TCP in end-to-end latency on preestablished connections, and does better for paradigms where connections must be constructed on the fly.

## 1 Introduction

The purpose of distributed parallel processing systems such as PVM, Mentat, Express, Linda, p4, and others, is to make a cluster of workstations appear to the user as a single multicomputer. These systems are typically software wrappers that facilitate the distribution of large problems among otherwise autonomous workstations, and that coordinate the communication necessary to achieve concurrent processing. This places a great deal of emphasis on the quality of the design and implementation of the data delivery system. In most cases, this delivery system is a network running the Internet protocol suite.

The cluster computing approach to parallel processing provides several advantages for both the research and production communities. Cluster computing offers a low cost alternative to expensive massively parallel processing (MPP) platforms, often allowing the utilization of existing resources and the gradual cumulation of systems. These systems can be expanded and improved through the purchase of additional resources and through the replacement of hardware components that limit performance. Further, public domain cluster computing software packages allow researchers to explore parallel processing options without substantial investments in new computing infrastructures.

However, a local area network is, in general, slower than the internal interconnection network of an MPP. More work must be done per communication event to amortize the communication cost over the computation. Consequently, the algorithms appropriate for implementation on a cluster of workstations are those with medium- and course-grain decompositions. Reducing latency and increasing throughput widens the category of algorithms appropriate for workstation clusters. One way to do this is to increase the capacity of the shared physical medium by replacing the ubiquitous 10 Mbit Ethernet with 100 Mbit LANs like FDDI, or by using switched media with aggregate

*This work was supported by Sandia Corporation under its Contract No. DE-AC04-94AL85000 with the United States Department of Energy, and CRADA No. 1136 between Sandia National Laboratories and AT&T Bell Labs.

bandwidth capacities on the order of a gigabit per second. This will help, but high latency and low throughput are also due to delays between the end user and the physical medium. The interface to the data delivery services, along with the protocols that comprise these delivery services, need attention.

The protocols used by distributed parallel processing systems should match the requirements of the applications they intend to support. Choosing a network protocol that provides excessive functionality can lead to unnecessary overhead that limits performance. Choosing one with inadequate functionality forces the missing services to be provided by the parallel processing system or application program. These makeshift extensions to existing protocols typically cannot be implemented as efficiently as protocols specifically designed to provide the appropriate services. Nonetheless, most cluster computing environments employ protocols from the Internet suite. PVM and Mentat, for example, both utilize their own reliable datagram protocols built above UDP, and PVM's alternate routing option uses TCP.

In this paper, we examine the Xpress Transfer Protocol (XTP) as the transport protocol for distributed parallel processing. We look at the communication characteristics of typical parallel applications, and use them to identify the performance and functionality requirements. We then discuss the features of XTP, TCP, and UDP that are relavent to providing efficient parallel processing. We run several different tests to measure end-to-end latency for all three protocols. Although protocol performance comparisons are largely implementation dependent, our results show that XTP's design and flexibility make it better suited than single paradigm solutions for meeting distributed parallel processing requirements.

## 2 Communication characteristics

Communication efficiency depends on several layers in the communication stack: the network protocols, the interface to the protocols, and the underlying network hardware. Shared media MAC layer protocols, like Ethernet, FDDI, and Token Ring, require a sender to gain exclusive access to the wire. Contention for the network increases with the number of processors onto which the problem is distributed, resulting in higher latency. The problem is exacerbated by parallel algorithms that progress in lock-step fashion and, in so doing, cause communication to take place simultaneously. Consequently, algorithms with medium- to course-grain decompositions are better able to amoritize the latency, and are therefore more appropriate for clusters over shared media LANs.

Switched media LANs (switched Ethernet, switched FDDI, and ATM), however, reduce or remove much of the contention, resulting in latency that is closer to the propogation time. Aggregate bandwidth, bounded in shared media LANs by the medium's data rate, becomes proportional to the number of channels in the switch. With these new technologies, cluster interconnects begin to resemble MPP interconnects in construction and performance. Even so, it has been shown that popular parallel processing systems, such as PVM, are still unable to fully exploit the performance of these high speed networks [6]. Emphasis must now be placed on optimizing the parallel processing systems and their transport protocols.

### 2.1 Parallel Algorithms

Several problem types have emerged as particularly appropriate for parallelization. Among these are iterative calculations over fields of values, such as stencil algorithms and multi-body dynamics simulations that use space-cell decompositions. A field of values is typically represented as a two- or three-dimensional matrix that is decomposed into cells and distributed among worker processes. The workers perform calculations, exchange results, and perform more calculations based on new information, continuing until some end condition is met. Typically, a cell communicates with the neighbors in it's "sphere of influence" to exchange the information necessary for the next round of calculations. A cell's immediate sphere includes eight neighbors in the 2 dimensional case, and 26 neighbors in the 3 dimensional case. A sphere of two cell-widths contains 24 and 124 neighbors for the 2D and 3D cases, respectively.

The amount of calculation by each processor in each iteration is proportional to the size of the cell, which is determined by the problem size and the number of processors onto which it is distributed. More processors implies more potential speedup, since the amount of computation per iteration per processor decreases with the size of the cell, leaving the communication phase as the limiting factor. In cluster computing, the communication costs can be orders of magnitude greater than for MPP machines. This argues for a large cell size to amortize communication costs over the longer calculation phases. But this limits speedup. The tradeoff can be shifted only by reducing the cost of the communication phase.

## 2.2 Communication topologies

Many algorithms have static communication topologies. For instance, in stencil algorithms, a vast majority of the communication takes place between processors that contain neighboring cells. Preestablishing connections between these communicants is certainly more efficient than connect-on-the-fly connection establishment, where connections are set up and torn down for each communication. However, some algorithms progress through several different phases during which the communication topologies are very different. In these algorithms, the penalty of preestablishing connections must be incurred not just once, but each time the program enters a phase that calls for a new topology.

Predetermining static communication topologies is impossible for some classes of applications. In the common bag-of-tasks paradigm, for instance, communicant pairs are not determined until runtime, when a task is assigned to a processor or set of processors. Using preestablished connections would require a fully general topology with connections between all possible pairs of processes. This scheme does not scale well, especially since connection oriented protocols often must incur the overhead of maintaining a separate communication record for each connection. Even worse, most protocol implementations typically treat an open connection as a limited resource, and impose a hard limit on the number that can be simultaneously maintained. Fully general topologies, and stencil algorithms that operate on cells that contain a large "sphere of influence," can push this hard limit. When the limit is reached, a connection must be closed each time a new one is required, resulting in performance no better than connect-on-the-fly connection establishment.

In connection oriented protocols, such as TCP, the cost of establishing a connection on the fly is often significant, especially when the amount of data being sent per communication is small, and when communication happens frequently. Connectionless protocols, such as UDP, avoid the overhead of explicit connection. However, UDP does not export enough functionality (e.g. reliability) to be suitable for distributed computing.

## 3 Protocol issues

Protocol implementors and protocol designers must do their parts to reduce latency. The implementor must limit the amount of processing required per packet. The designer must limit the number of packets per service operation and provide functionality that is appropriate to the higher layer protocol or application. Distributed parallel computing applications need several different types of communication paradigms. A protocol successful in distributed parallel processing must be flexible enough to support these paradigms efficiently. In this section, we discuss the features of TCP, UDP, and XTP, that are relevant to providing appropriate service.

## 3.1 TCP and UDP – the Internet suite

TCP and UDP over IP have served well as the foundations of the Internet, especially for applications that fit into one of the two paradigms offered. One-shot unreliable data delivery is provided by UDP. This is appropriate when the application or higher level protocol produces a limited amount of data sequentially unrelated to any other UDP datagram, and when recovery of lost data is unnecessary. Otherwise, some protocol above UDP must provide complete in-order delivery. TCP employs the internal mechanisms to track data delivery and to recover lost data. In this way, TCP provides the notion of a reliable stream of data in which order is relative and important. TCP does not provide message boundary markers.

TCP uses a three-way handshake to establish a connection [4], where initial sequence numbers for both directions of flow are exchanged. The three-way handshake guards against connection hazards caused by duplicate packets. Once open, the two communicants exchange data reliably. Graceful close of a connection requires each side to know that the other has received all data sent.

The Berkeley and System V Unix-based implementations of TCP [5, 7], through the socket and TLI interfaces, separate the connection establishment, data transfer, and connection closing into

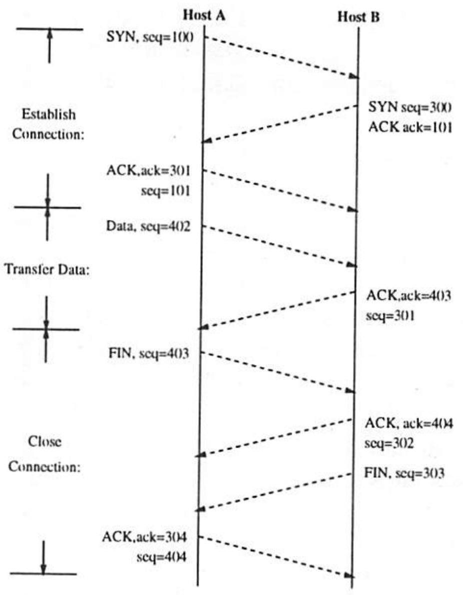


Figure 1: TCP data transfer

three stages. Since the TCP specification does not prevent the connection close handshake from piggybacking on the connection setup handshake, in theory only three segments are needed to reliably deliver TCP data [3, 2]. In practice, however, at least nine packets are used, as shown in Figure 1.

Because of its popularity and longevity, TCP is a target for implementation optimizations. Van Jacobson has observed that the predictability of future packets, based on previously received packets, is high enough to allow significant performance tuning of TCP (some of the observations made by Jacobson and others to tune TCP and UDP are chronicled in [8]). Further, fast-path studies of TCP [1] suggest that TCP packets can be parsed in about 200 instructions. Less can be done with issues that involve the design of the protocol. Once a protocol is widely available, the packet use and functional offerings are already decided and difficult to alter.

### 3.2 The Xpress Transfer Protocol

The Xpress Transfer Protocol (XTP) [11, 9] provides a transport layer service without dictating the communication paradigm or the delivery characteristics that qualify the paradigm. Communication paradigms are patterns of packet exchanges. Qualifiers on a paradigm, such as reliable or unreliable delivery, indicate how the endpoints (called *contexts* in XTP) act under various conditions, such as lost data. Central to the design of XTP is the notion that protocol service flexibility is essential for providing support for a wide range of applications. XTP allows considerable control over the pattern of packet exchanges and, through option bits carried in the packet header, provides control over the way these packets and their contents are handled.

There are three main packet types in XTP: FIRST packets, DATA packets, and CNTL packets. A FIRST packet is the first packet of an association. It is only used once if not lost, and carries addressing information and optional data. A DATA packet carries only data. A CNTL packet carries context state information, including flow and error control notifications.

Figure 2 shows several of the option bits of the header. The SREQ (status request bit) causes the destination context to respond with a CNTL packet that carries state information, including the current status of the data transfer. This bit allows the transmitter to request acknowledgements at times appropriate to the paradigm.

Several mode bits qualify the association between endpoints. The FASTNAK (fast negative acknowledgement) bit instructs the receiver to generate a CNTL packet in response to a DATA packet that is received out of sequence number order. The NOFLOW (no flow control) bit indicates that transmitted data cannot be throttled and that allocation information sent back to the transmitter

| Bit | Description |
|---|---|
| NOCHECK | Turns off checksum over all but header |
| NOERR | Turns off error control |
| MULTI | Indicates multicast association |
| RES | Indicates conservative allocation policy |
| SORT | Indicates the presence of a priority value |
| NOFLOW | Turns off flow control |
| FASTNAK | Indicates aggressive error reporting |
| SREQ | Status request immediately |
| DREQ | Status request after data has been delivered |
| RCLOSE | Signals close of reader process |
| WCLOSE | Signals close of writer process |
| EOM | Marks end of message |
| END | Indicates end of association |
| BTAG | Indicates the presence of out-of-band data |

Figure 2: XTP option bits

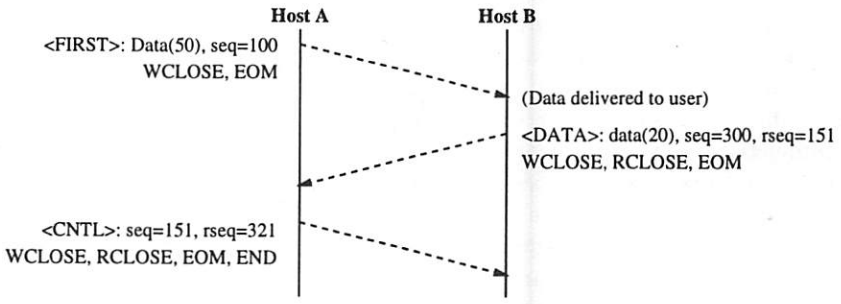


Figure 3: XTP reliable transaction

will be ignored. The NOERR (no error control) bit instructs the receiver to ignore gaps in the data stream.

Other option bits mark the data stream. The EOM (end of message) bit marks the end of a logical unit of data. The transmitter sets the WCLOSE (writer closed) bit to indicate that no new data will be sent. This marks the end of the data stream—only retransmitted data may be sent thereafter. The RCLOSE (reader closed) bit indicates that the context setting this bit will accept no more data packets on its incoming data stream. It is used to acknowledge the receipt of the WCLOSE bit, and implies that all data in the stream has been received. The END (end of association) bit indicates that the sender has closed the association and that the receiver should do so as well.

The FIRST packet's addressing information is used by the destination host to match the packet with the appropriate listening context. Since this FIRST packet may also carry data, overhead for unreliable data delivery is minimal. (Even in assured delivery paradigms, XTP allows the application to send data aggressively before it confirms association establishment.) A reliable datagram is constructed by sending data and an SREQ in the FIRST packet, then waiting until the CNTL packet is returned. A CNTL packet from the initiator to the receiver closes the association.

A transaction is built similarly, as illustrated in Figure 3. A FIRST packet with data (and as many DATA packets as required) serves as a request. The end of the request is marked by setting the EOM and WCLOSE bits in the last packet of the message. The response, in as many DATA packets as are necessary, implicitly acknowledges the request. If the transaction is not reliable, the last packet of the response carries the END bit and closes the association. If the transaction is reliable, the transmitter closes the connection with a control packet carrying the END bit.

Option bits, in concert with packet exchange patterns, are the mechanisms that allow XTP to provide services appropriate to application requirements.

## 4 Performance

The Transport Layer Interface (TLI) is designed to be System V Unix's standard interface to transport layer protocols. Mentat Inc., of Los Angeles, California[1], has developed a kernel implementation of XTP Version 3.7, called MXTP [10], with a TLI interface. To characterize the performance of XTP, we ran several different tests using MXTP and the implementations of TCP and UDP (also with TLI interfaces) that come standard with SunOS 5.3 (Solaris). The timing tests were run between two Sun SparcStation 10 Model 50 workstations containing Viking processor chips. The machines communicated through a dedicated channel on a 100 Mbps DEC FDDI Gigaswitch.

We tested five different protocols—XTP, TCP, UDP, R-UDP, and T-UDP—for three different communication paradigms. R-UDP is a simple approximation of what a reliable datagram protocol should do. We built R-UDP over UDP using a three-way positive acknowledgement packet exchange with two timers to catch lost packets. This three-way handshake ensures that the transmitter and the receiver are aware of each other's states as quickly as possible. If a two-way exchange is used to construct reliable delivery, the receiver would need a dally timer to catch duplicate data packets from the transmitter in case the acknowledgement got lost. The third packet, from the transmitter back to the receiver, assures the receiver that the transmitter knows the data was received. T-UDP is a similarly constructed transaction primitive built over UDP using two timers, one for the request and one for the response, and a single acknowledgement of the response. The response acts as an acknowledgement for the request. Since no packets happened to be dropped in any of the R-UDP or T-UDP tests, our measurements include the overhead of providing the mechanisms for recovery of lost data, but not the overhead of actually recovering any lost data.

We refer to the three communication paradigms as "Preestablished," "One-Shot," and "Transaction." Preestablished measures the one-way end-to-end latency of a message sent on an already established connection. One-shot measures the end-to-end one-way latency, including the time for connection setup and teardown, of a single message delivered reliably. This test is designed to estimate the communication performance that can be achieved when connections cannot be held open throughout the computation. Transaction measures the performance of the protocols performing with request-response behavior similar to RPC.

We took measurements of these paradigms for message sizes of 4, 16, 64, 256, 512, 1024, 2048, and 4096 bytes. We chose these message sizes to reflect the size of data that is appropriate for various parallel algorithms. Our sample size was twenty separate timings per data point. Each timing run consisted of 50 iterations of the experiment; if the experiments used roundtrip times to measure the latency, the total time was divided by 100, otherwise the total time was divided by 50. As a consequence, 1000 samples were taken for each point plotted. The results we show estimate the mean within plus or minus 1% at a confidence level of 95%. For each of the experiments we established a steady state before taking measurements to avoid various artifacts such as ARP lookups.

The results of the Preestablished experiments are shown in Figure 4. For each protocol we measured the time to send and fully receive the data, giving true end-to-end latency. Analysis of the data shows that there is an overhead component and a per-byte component to each protocol's latency, and that the per-byte component for each protocol is essentially the same. Fitting a line to the data shows the overhead as the y-intercept and the per-byte cost as the slope:

$$\begin{aligned}
\text{TCP Overhead} &= 0.650137 \text{ ms, Per-byte cost} = 0.344\ \mu\text{s}\\
\text{XTP Overhead} &= 0.649867 \text{ ms, Per-byte cost} = 0.344\ \mu\text{s}\\
\text{RUDP Overhead} &= 1.851929 \text{ ms, Per-byte cost} = 0.347\ \mu\text{s}\\
\text{UDP Overhead} &= 0.554301 \text{ ms, Per-byte cost} = 0.334\ \mu\text{s}
\end{aligned}$$

The difference between the UDP and R-UDP curves, about 1.3 ms, is the cost of adding two acknowledgement packets to make R-UDP reliable. This cost is constant since the acknowledgement packets are a fixed size. TCP and XTP add about a tenth of a millisecond to the overhead of a UDP packet. This is likely due to the state information that must be updated with the arrival of new packets, and an amortized cost of data acknowledgement.

[1]Mentat Inc. is unrelated to the University of Virginia's parallel processing system, also called Mentat.

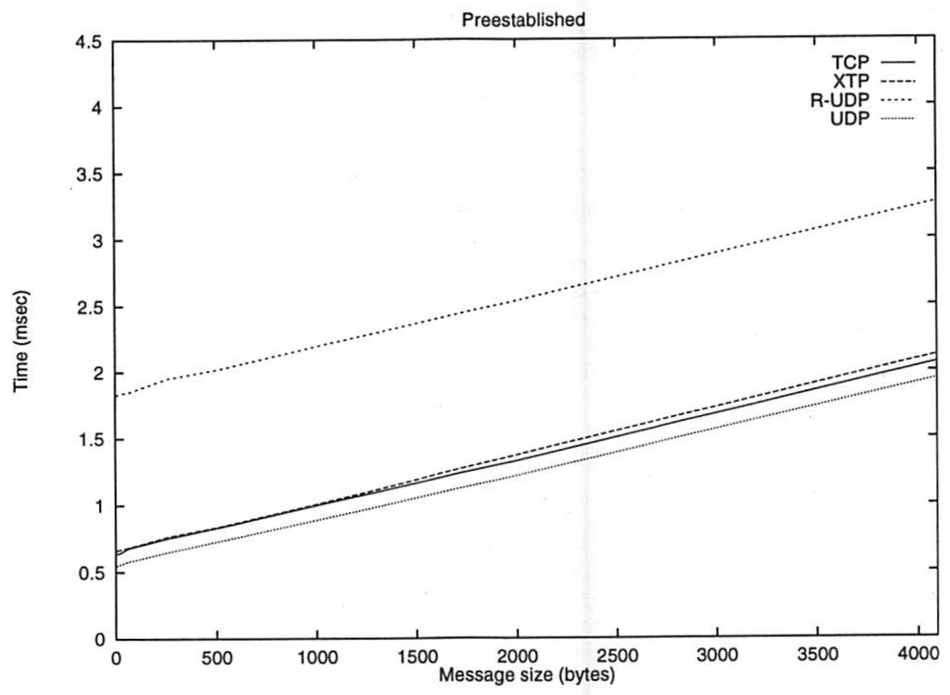


Figure 4: Time for one-way send on preestablished connection

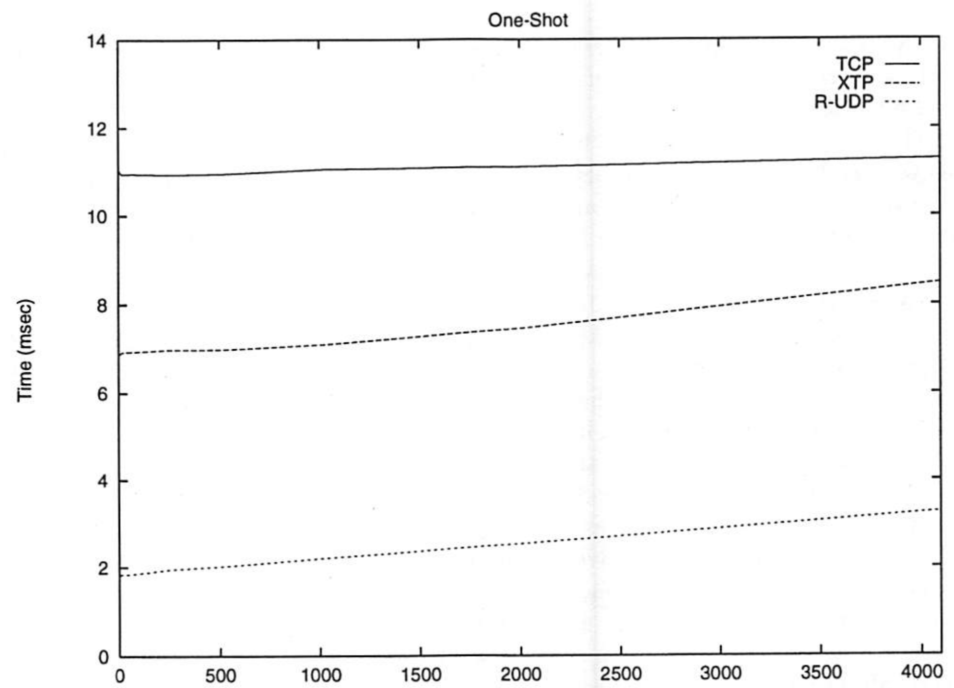


Figure 5: Time for one-way send, including connection setup and teardown

```
Sender
Fill in a call structure
Connect using this call structure
Send data
Send and orderly release
Block on a receive; when it fails (we expect it to),
        recieve the disconnect message

Receiver
Listen on the previously opened file descriptor
When a connect request arrives, open another file descriptor
Bind to it
Accept the connection on the new file descriptor
While there is data, receive
When the receive fails (we expect it to),
        receive the orderly release message
Send a disconnect message
```

Figure 6: Pseudo-code for TCP One-Shot

```
Sender
Fill in a call structure
Connect using this call structure and a data buffer
Send and orderly release
Block on a receive; when it fails (we expect it to),
        recieve the disconnect message

Receiver
(Same as in Figure 6)
```

Figure 7: Pseudo-code for XTP One-Shot

The One-Shot experiments, shown in Figure 5, expose the costs of connecting, sending, and disconnecting during a single communication with an arbitrary receiver. The open call, which allocates a file descriptor, is done only once; a call structure is filled in on the fly with the intended receiver. This file descriptor can then be reused for communication with another receiver.

We show only the reliable protocols in this graph. R-UDP does not have an explicit connect call, so the times shown by it's curve are the same as in the Preestablished graph. This protocol is the least expensive because it has the least to do. TCP and XTP both have explicit connect calls, but XTP allows the user to send data in the FIRST packet. This reduces the number of packets necessary, which is reflected in the timings. XTP should need only two packets and a dally timer (built into the protocol), or three packets without a dally timer, to reliably send one piece of data, but MXTP uses five: the FIRST packet and it's acknowledgement, then a three-way handshake to disconnect. This is more the fault of the TLI interface than of MXTP. TLI does not allow a connect, send, and disconnect to be combined in a single primitive. TCP does not couple the connect and send primitives; as shown in Figure 1 the connect, send, and disconnect phases are separate.

The psuedo-code for the TCP One-Shot data transfer is shown in Figure 6. We use the failure of the normal receive to detect a release message. This is necessary because TCP has no notion of a message bounary, so either the receiver must know the size of the message *a priori*, or the release message must be used to mark the end of transmission. Also notice that a disconnect message is used. This is normally used to abort a connection. By the time the disconnect function is called, the receiver has gotten all the outstanding data. Further, we could not call the connect function immediately after an orderly release because of some delay imposed by TCP to ensure all packets for this connection are received. Timings with the two-way orderly release showed times in the one-half

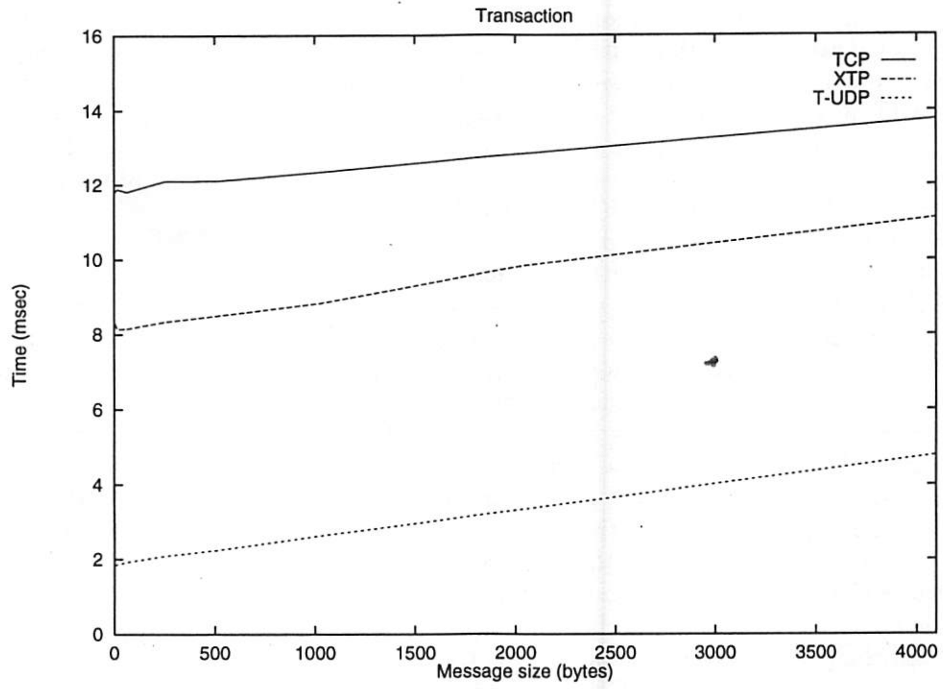


Figure 8: Time per transaction, including connection setup and teardown

to one second range. The psuedo-code for the sender for the XTP One-Shot data transfer is shown in Figure 7.

Figure 5 clearly shows that the XTP connect with data, done via a FIRST packet, helps reduce the one-way latency. According to [10], the number of packets used for this One-Shot send is five: the FIRST packet with DATA from sender to receiver, a CNTL packet in reply, and a three way CNTL packet exchange to close the connection. There are some efficiencies missing in the MXTP implementation, such as piggybacking the beginning of the connection disconnect on the FIRST packet, but the TLI interface does not expose that level of control.

The Transaction results are shown in Figure 8. As with the One-Shot results, these numbers reflect the cost of connecting, reliably executing a transaction, and disconnecting so that the file descriptor can be reused. The response messages in this experiment are the same size as the request messages; we acknowledge that this will rarely be the case, but there is no standard ratio of response size to request size. Also, as with the One-Shot results, reliable UDP is the least expensive because it uses only three packets and two timers for assuring reliable delivery. The overheads for XTP and UDP are roughly four and six times that of T-UDP, respectively; there are more packet exchanges taking place, and the protocol state machines for XTP and TCP are much more complex than that of T-UDP. MXTP's t_connect() uses the two call structure pointers to send the request and receive the response; the transaction, except for the disconnect, is done in one function call. After this a three-way exchange closes the connection. Again, more efficiency can be achieved by coupling some of the connection release indications with the packet exchange used to send the request and response, as shown in Figure 3.

## 5 Conclusions

Comparing the performance of various protocols, even on the same platform with the same interface, must be done cautiously. So much depends on implementation decisions and programming skill that the inherent advantages of one protocol over another can be lost. Nonetheless, we endeavor to examine TCP, UDP, and XTP for several paradigms important for cluster computing. Implementation differences aside, a protocol must be flexible and efficient in order to serve distributed parallel processing environments. High latency is extremely damaging to parallel processing performance. Well designed and well implemented protocols are essential in any environment.

XTP is designed to handle common virtual circuits as well as datagrams. It provides reliability as an orthogonal attribute; with the packet types and options bits in XTP one can construct reliable datagrams, unreliable virtual circuits, or any number of combinations of attributes and paradigms. In theory XTP provides the tools for constructing communication primitives appropriate for many types of environments. We obtained a commercial implementation of XTP from Mentat, Inc. and instrumented several experiments based on paradigms that arise in parallel processing in order to compare this XTP with the native TCP and UDP.

We make several observations based on the results of these experiments and our experience running them. XTP compares favorably with TCP for one-way end-to-end latency on a preestablished connection. For the situations where a paradigm such as reliable datagrams or reliable transactions had to be constructed, TCP performed worse than XTP. If we assume that the results from the latency tests over preestablished connections suggest that the protocols are comparably implemented, XTP's advantage must be due to reduced packet exchanges. For the one-shot reliable datagram and the reliable transaction, XTP carries data in the FIRST packet while TCP requires the three-way connect handshake before data can be exchanged. MXTP exposes these aspects of XTP through its t_connect() call. Both TCP and XTP, through the TLI interface, required a fairly extensive sequence of calls for reliably closing the connection, and even then the orderly release for TCP caused delays of as much as a second before releasing the resources. Our experience in working with MXTP suggests that it either can not or does not expose the full flexibility of XTP to the user. Some of this is design decision, some is due to the TLI interface. At any rate, more economy could have been gained from XTP in both the reliable datagram and reliable transaction.

In spite of the advantages of XTP over TCP in the experiments where connections where established and released on the fly, the clear winner in reducing latency is a preestablished connection. When the communication topology is static enough to allow this, a preestablished connection provides the lowest latency of reliable protocols. For connect-on-the-fly communication, it seems that our R-UDP and T-UDP solutions are faster than TCP or XTP; we caution that these sample protocols hold virtually no state information or do other protocol processing. Still, these constructed protocols show what is possible when a minimum number of packets are used to construct a particular communication paradigm.

**Acknowledgments**

The authors wish to thank John Fenton, of Mentat, Inc., for the valuable technical assistance and insight, and Bruce Carneal, also of Mentat, Inc., for providing MXTP for evaluation. We also thank Tuesday Armijo, Tim Cody, and Alden Jackson of Sandia.

## References

[1] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[2] D. E. Comer. *Internetworking with TCP/IP, Vol: I, Principles, Protocols, and Architecture: Second Edition.* Prentice Hall, Edgewood Cliffs, New Jersey, 1991.

[3] W. A. Doeringer, D. Dykeman, M. Kaiserswerth, B. W. Meister, and H. Rudin. A Survey of Light-Weight Transport Protocols for High-Speed Networks. *IEEE Transactions on Communications*, 38(11):2025–2039, November 1990.

[4] J. Postel (editor). Transmission Control Protocol, rfc-793, September 1981.

[5] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, Reading, Massachusetts, 1989.

[6] M. J. Lewis and R. E. Cline Jr. PVM Communication Performance in Switched FDDI Heterogeneous Distributed Computing Environments. *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 13–19, October 1993.

[7] M. Padovano. *Networking Applications on UNIX System V Release 4*. Prentice Hall, Edgewood Cliffs, New Jersey, 1993.

[8] C. Partridge and S. Pink. A Faster UDP. *IEEE/ACM Transactions on Networking*, 1(4):429–440, August 1993.

[9] W. T. Strayer, B. J. Dempsey, and A. C. Weaver. *XTP: The Xpress Transfer Protocol.* Addison-Wesley, Reading, Massachusetts, 1993.

[10] Mentat XTP Internals Manual, Mentat, Inc., Los Angeles, California, 1994.

[11] XTP Protocol Definition, Revision 3.6. Technical Report PEI 92-10, Protocol Engines, Inc., January 1992.

# ViewStation Applications: Intelligent Video Processing Over a Broadband Local Area Network *

Christopher J. Lindblad, David J. Wetherall, William F. Stasior, Joel F. Adam, Henry H. Houh, Mike Ismert, David R. Bacher, Brent M. Phillips, and David L. Tennenhouse †

*Telemedia Networks and Systems Group*
*Laboratory for Computer Science*
*Massachusetts Institute of Techology*

## Abstract

This paper describes applications built on the ViewStation, a distributed multimedia system based on Unix workstations and a gigabit per second local area network. A key tenet of the ViewStation project is the delivery of media data not just to the desktop but all the way to the application program. We have written applications that directly process live video to provide more responsive human-computer interaction. We have also developed applications to explore the potential of media processing to support content-based retrieval of pre-recorded television broadcasts. These applications perform intelligent processing on video, as well as straightforward presentation. They demonstrate the utility of network-based multimedia systems that deliver audio and video data all the way to the application. The network requirements of the applications are a combination of bursty transfers and periodic packet-trains.

## 1 Introduction

The ViewStation project [3] integrates the technologies of broadband networking and distributed computing with those of digital video to produce systems for video-intensive computing. The ViewStation platform is composed of a set of programmable digital video processing devices connected together in a personal local area network.

The project focuses on getting real-time data such as voice and video from the network all the way to the application. ViewStation hardware is centered around the VuNet [5], a prototype desk area network to link the display, camera, central processor, and wide-area network components. Since the ViewStation takes a software-intensive approach to multimedia, the VuNet and custom multimedia hardware were designed to provide efficient support for software-driven handling of multimedia streams.

This paper describes applications built with the VuSystem [1], the programming system that provides application support for the ViewStation project. The system provides simple scheduling and resource management functions to allow intelligent media-processing applications to run on workstations not specifically designed for multimedia. Examples of such applications include The Room Monitor, The Whiteboard Recorder, The Video Rover, The News Browser, The Joke Browser, and The Sports Highlight Browser.

*This research was supported by the Advanced Research Projects Agency of the Department of Defense, monitored by the United States Air Force (AFSC, Rome Laboratory) under contract No. F30602-92-C-0019, and by a grant from Nynex.

†The authors can be reached at: MIT Laboratory for Computer Science, Room 504, 545 Technology Square, Cambridge, MA 02139; Tel: +1 617 253 6042; Email: cjl@lcs.mit.edu.

## 2 Principles

The ViewStation architecture embodies a software-oriented approach to the support of interactive media-based applications. Starting from the premise that the raw media data, e.g., the video pixels themselves, must eventually be made accessible to the application, we have derived a set of architectural guidelines for the design of media processing environments.

*Video information must be accessible to and manipulatable by the application.* We want to enable a new wave of media applications in which the computer is an active participant. These applications analyze their audio and video data input and take actions based upon the analysis.

*A software approach, preserving scalability and graceful degradation, is called for.* Software-oriented applications adapt to the resources of the execution platform and to the dynamic load applied by the mix of concurrently active applications.

*Perceptual time is the domain of interest for interactive applications.* Real-time software environments are discrete systems that approximate real world time to the finest level of temporal granularity that can be achieved by the available technology. However, we need only approximate real world time at a granularity that satisfies human perceptions, a granularity we refer to as perceptual time.

*The resultant software and media substrate must ride the technology curve.* To be of long-term relevance, media architecture and software strategies must leverage rapid change in the underlying technologies, especially processors and memories. Each generation of technology affords increased sophistication in the media processing performed by the applications.

## 3 Other Work

In general, the design philosophy behind the ViewStation differs from other network-based multimedia approaches that are more hardware based, such as Pandora [15], a special purpose hardware sub-system which can be controlled by a workstation. Pandora's box performs video and audio capture as well as mixing in dedicated hardware attached to the workstation and under workstation control. The output of the box goes directly to the workstation display.

The Pandora approach fits into the dedicated hardware style of media delivery. Very specialized hardware is built which allows decent performance, but prevents taking advantage of gains in performance elsewhere in the system.

The Medusa project [4] has improved on the design of Pandora in the areas of portability, security, programmibility, and the support of multiple streams. Its design is similar to that of the ViewStation. It connects the network to the workstation, allows video to pass all the way to applications, and provides for the development of software *agents* to assist in collaboration. It is built upon simple, lightweight connections, so that it is easy to build large, understandable groups of modules.

Other approaches have attacked multimedia systems from the network aspect. The Atomic approach [13] uses a networking chip designed for multiprocessing. The network chips are used to interconnect the various subsystems such as a display, camera, DSP, and monitor. The Atomic approach uses specialized network hardware as a multimedia workstation's internal network.

The approach of the University of Cambridge [14] replaces the traditional bus structure of the workstation with an ATM network. This network interconnects a variety of multimedia devices on a fast communications substrate internal to the workstation, and resides between the processor and its memory hierarchy. Our VuNet approach, while similar, chooses not to intercede into this memory hierarchy, and thus places the ATM network boundary at the edge of the memory subsystem.

## 4 The VuNet

The VuNet is a gigabit-per-second network using Asynchronous Transfer Mode technology (ATM) [9, 11] that interconnects general-purpose workstations and multimedia devices. The VuNet is aimed at the desk-area and local-area environments.

A key goal motivating the design of the VuNet hardware has been simplicity. Our goal is to design network hardware that is easy to build with off-the-shelf VLSI components. More sophisticated

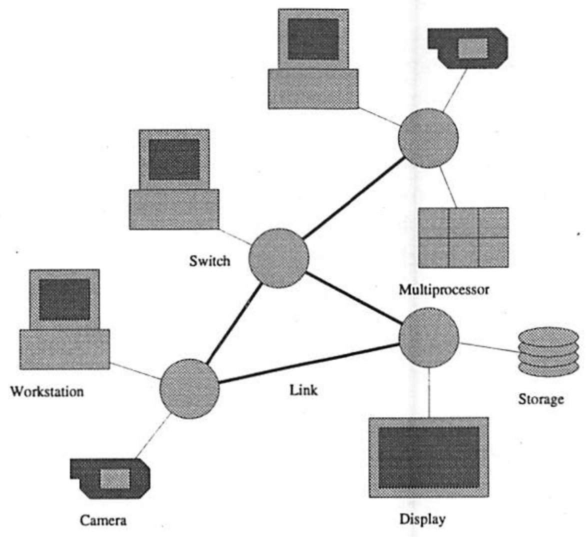


Figure 1: The VuNet.

network functions, including multicast, backpressure, support for ATM adaptation layers, and service classes, were pushed to the edge of the network where they become the responsibility of the clients. We believe that a local environment such as the ViewStation does not need sophisticated congestion-control functions. It can be effectively served by a switch fabric having a limited number of access ports and an internal speed that is greater than that of the clients.

The VuNet is based on the interconnection of two types of components: switches and links. The switches provide ports through which clients connect to the network and execute cell switching. The links interconnect the switches and implement cell relaying. With these two basic components, different network topologies are possible. Within a desk-area context, an office is equipped with a network switch which interconnects workstations and multimedia devices. Offices are then networked by connecting individual switches with links.

The VuNet switch consists of bidirectional ports with first-in first-out (FIFO) buffers which feed a crossbar matrix. The current version of the switch has four ports and has been operating reliably at a speed of 700 Mb/s. The switch has tested reliably at an internal data rate of 1.5 Gb/s.

High speed optical links provide inter-connection between VuNet switches. Cells are routed hop-by-hop as they pass through the links. Links contain header lookup tables that map VCIs to VCIs and switch ports. Connection management is performed through special control cells on a particular VCI. Control cells, received by a link on this special VCI, program the link's header lookup tables.

Each workstation in the VuNet is responsible for opening, maintaining, and closing its own connections. This can be done in a "wormhole" fashion by way of ATM control cells embedded in the data stream. The allocation algorithms in the connection daemon prevent nodes from stealing other nodes' connections. Processes are also run in the background which verify link tables and refresh connections if necessary, such as in the case where a switch has been restarted.

## 4.1 VuNet Clients

Three types of clients have been integrated into the VuNet: workstations, specialized multimedia devices, and inter-network interfaces. General-purpose workstations are connected through a host interface. Specialized multimedia devices, including video capture boards and an image processing system, are connected directly to switch ports. Inter-network bridges connect the VuNet to local-area and wide-area ATM networks.

The VuNet interface between client and switch was designed to be simple, allowing clients to be easily built. This follows the software intensive philosophy, which emphasizes a simple, flexible hardware substrate, and pushes complex functionality into workstation-based software.

The VuNet host interface is a simple DMA interface. Incoming cells are packed along with other relevant information into processor memory, where a kernel device driver reassembles packets and delivers them to the proper application. Outbound cells are packed by the driver in main memory with routing information and timing information (outgoing cells can be paced on a per-cell basis), and are transferred to the interface using block DMA transfers.

From the application perspective, media data is presented at the Unix socket interface. This provides a uniform software interface for all media data. Each media stream is mapped to its own ATM based virtual circuit. The VuNet kernel device driver implements the one-to-one mapping between application sockets and network virtual circuits. For incoming traffic, the device driver maps ATM Virtual Circuit Identifiers (VCIs) to their respective application socket interfaces. Throughput to the application has been measured at 40 Mb/s.

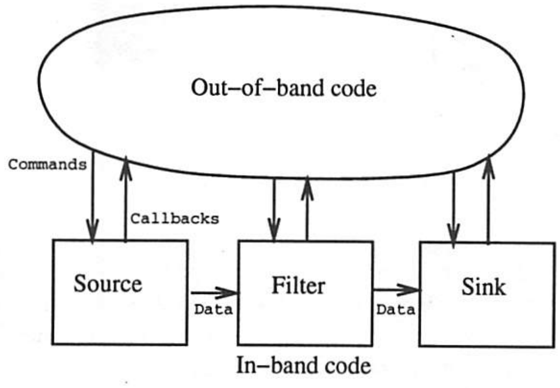


Figure 2: The structure of VuSystem applications.

## 5 The VuSystem

VuSystem applications are split into two partitions: one which does traditional *out-of-band* processing and one which does *in-band* processing (Figure 2). Out-of-band processing is the processing that performs event handling and other higher-order functions of a program. In-band processing is the processing performed on every video frame and audio fragment. In-band code is more elaborate in the VuSystem than in traditional multimedia systems [8, 10] because VuSystem applications perform more analysis of their input media data.

In the VuSystem, the in-band processing partition is arranged into processing *modules* that logically pass dynamically-typed data *payloads* though input and output *ports*. These in-band modules can be classified by the number of input and output ports they possess. The most common module classifications are sources, with no input ports and one output port; sinks, with one input port and no output ports; and filters with one or more input ports and one or more output ports.

The out-of-band partition is programmed in the Tool Command Language, or Tcl [12], an interpreted scripting language. Application code written in Tcl is responsible for creating and controlling the network of in-band media-processing modules, and controlling the graphical user-interface of the application. In-band modules are manipulated with *object commands*, and in-band events are handled with asynchronous *callbacks*.

The VuSystem is implemented on Unix workstations as a shell program that interprets an extended version of Tcl. All out-of-band code, including all user-interface code, is written as Tcl scripts. In-band modules are implemented as C++ classes and are linked into the shell. Simple applications that use the default set of in-band modules are written as Tcl scripts. More complicated applications add additional in-band modules to the default set.

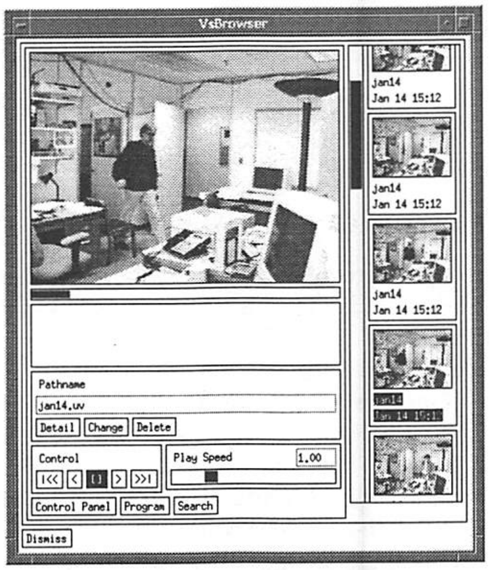
VuSystem programs have a *media-flow* architecture: code that directly processes temporally sensitive data is divided into processing *modules* arranged in data processing *pipelines*. This architecture is similar to that of some visualization systems [16, 17], but is unique in that all data is held in dynamically-typed time-stamped *payloads*, and programs can be reconfigured while they run. Timestamps allow for media synchronization. Dynamic typing and reconfiguration allows programs to change their behavior based on the data being fed to them.

## 6 Applications That Process Live Video

We have found that the ViewStation provides a good platform for the investigation of concrete ways that computers may become more responsive to their human users. We are developing a prototype "Computerized Office Multimedia Assistant" (COMMA), that assists its user by performing various tasks that require the analysis of live video.

We have developed a library of vision service modules as a foundation for COMMA applications. Example modules include a *change detector*, which accepts two input images and outputs a binary image which shows which pixels differ on the two input images; a *motion detector*, which detects and localizes motion in a video stream by outputting a true value for those pixels which correspond to moving objects in the scene; and a *stationary filter*, which is effectively the converse of the motion detector.

The vision service modules communicate with the higher level scripting language by signaling events or callbacks to the Tcl layer. For example, a face recognition service would be implemented as a filter that calls a Tcl subroutine whenever a model face appears in the video stream. The Tcl program would then determine how to use this information.



Pathname: jan14.uv

Play Speed: 1.00

Figure 3: Browsing the output of the Room Monitor.

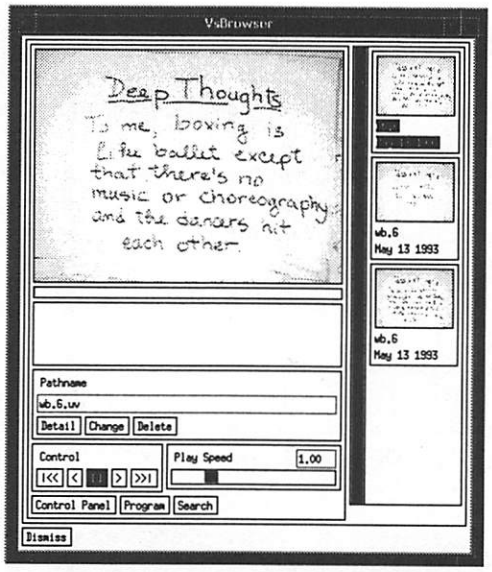
### 6.1 The Room Monitor

*The Room Monitor* processes periodic video data[1] from a stationary camera in a room. It processes

[1] In cases where the average data rate equals the peak channel capacity, applications pass data continuously. However, on the VuNet the data rate is less than the channel capacity, so typically connections pass data in periodicly recurring packet-trains.

the live video to determine if the room is occupied or empty, and records video frames only when activity is detected above some threshold. It produces a series of video clips that summarize the activity in the room. A video browser (Figure 3) is used to view the segments. The video clips allow the user to check who was in the room and when.

With additional processing, it is possible to extract people from the clips, using motion segmentation techniques. By processing these extractions through a recognition module, it might be possible to automatically figure out who the people are. The program might also be able to determine how many people are in the room, and what changes were made to the room during each visit.


Deep Thoughts

To me, boxing is like ballet except that there's no music or choreography and the dancers hit each other.

Pathname: wb.6.uv

wb.6 May 13 1993

wb.6 May 13 1993

Play Speed: 1.00

Figure 4: Browsing the output of the Whiteboard Recorder.

## 6.2 The Whiteboard Recorder

We have also written *The Whiteboard Recorder*, an application that keeps a history of changes to an office whiteboard. It works by taking video from a stationary camera aimed at the whiteboard and filtering it. By following a simple set of rules, the filtering distills the video into a minimum set of images. A browser (Figure 4) can be used to view the saved images.

The whiteboard recorder uses motion analysis to distinguish between the person writing on the board and the writing itself. Live video captured from a fixed camera is processed so that transient image features are filtered out, and only relatively stationary features are retained. This filters out people passing between the camera and the whiteboard. The filtered images are reconstructions derived from many partial images of the whiteboard — it is possible to capture an unobstructed view of the board from a sequence of input images where the board is partially occluded at all times.

From these relatively stationary images of the whiteboard, the program next distinguishes changes to the whiteboard due to writing from changes due to erasing. The system saves away images that represent "peaks" in the information written on the board.

## 6.3 The Video Rover

We have built *The Video Rover* to explore additional uses, such as remote sensing and telepresence, for live video networking. The Video Rover is an untethered vehicle that communicates with the VuNet using wireless transmission. Built from off-the-shelf electronics, the rover can be driven from a computer console on the VuNet.

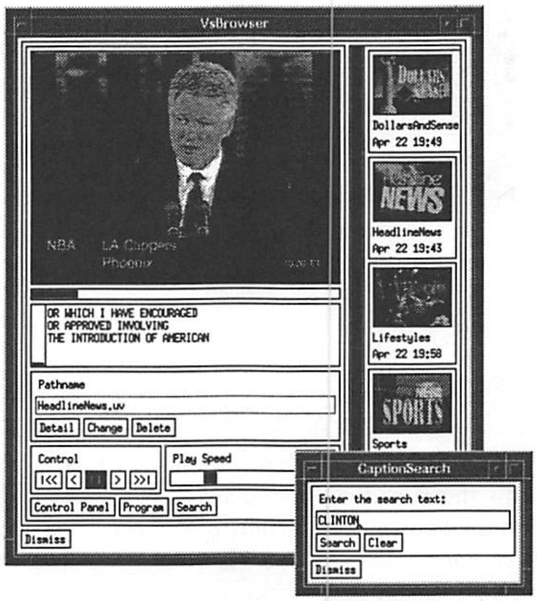
The Video Rover consists of a video camera mounted on a small remote-controlled car and a wireless video link. We have integrated the radio controller with the VuSystem so that the car can be controlled from a VuSystem application, with the rover returning video feedback to driver of the vehicle.

## 7 Content-Based Processing Of Television Programs

We expect a significant portion of data carried on wide-area broadband networks to be pre-produced media similar to broadcast television. We have used the ViewStation to explore the potential of media processing applications to support content-based retrieval of pre-recorded television broadcasts.

We have developed content-based media browsers that use textual annotations that represent recognizable events in the video stream. These annotations are analyzed and processed to create higher level representations that may be meaningful to a human user. Finally, these representations are matched against user queries to generate an interactive presentation in the form of a browsable set of relevant video clips.

Annotations are generated through the recognition of audio or video cues from the media stream, or by the extraction of ancilliary information included in the stream, such as closed captions. The News Browser, The Joke Browser, and The Sports Browser are built on increased levels of processing of these annotations.


OR WHICH I HAVE ENCOURAGED
OR APPROVED INVOLVING
THE INTRODUCTION OF AMERICAN

Pathname: HeadlineNews.uv

Enter the search text: CLINTON

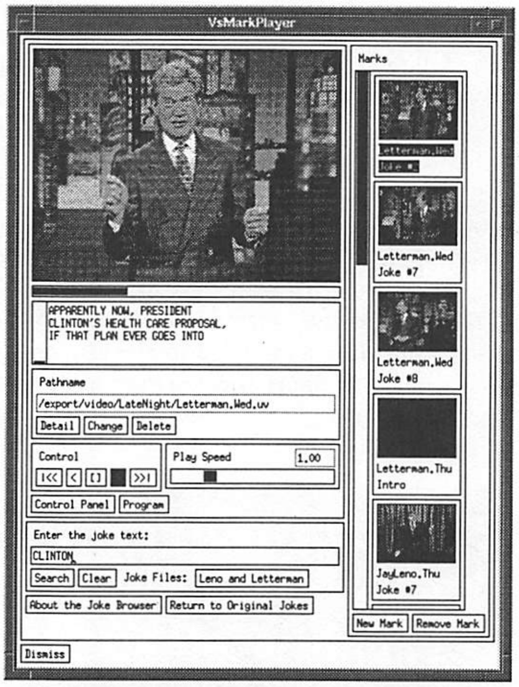
Figure 5: The News Browser.

### 7.1 The News Browser

*The News Browser* provides interactive access to a simple database of broadcast television news articles. Live television news programs such as CNN Headline News are automatically captured to disk at regular intervals. The stories are viewed with a video browser program (Figure 5).

News stories that are closed-captioned can be retrieved based on their content. Many broadcast television programs are closed-captioned for the hearing-imparied. Closed-captions provide a text translation of the audio component of the program — a significant amount of information. We wrote closed-caption capturing code for the Vidboard [6]. The caption information is extracted from the digitized video signal and converted into a common format so that modules capable of processing it can be constructed.

The news browser makes direct use of the closed-captioned annotations. A text search specification supplied by the user causes the browser to jump to stories with captions that match.



```
APPARENTLY NOW, PRESIDENT
CLINTON'S HEALTH CARE PROPOSAL,
IF THAT PLAN EVER GOES INTO
```

Pathname: /export/video/LateNight/Letterman.Wed.uv

Enter the joke text: CLINTON

Figure 6: The Joke Browser.

## 7.2 The Joke Browser

We have developed *The Joke Browser*, which further demonstrates the potential of content-based media processing using closed-captions [2]. It records late-night talk show monologues, and segments them into jokes by processing the closed-captioned text. A special browser program (Figure 6) is queried to select all the jokes on a certain topic that have been made in the last week.

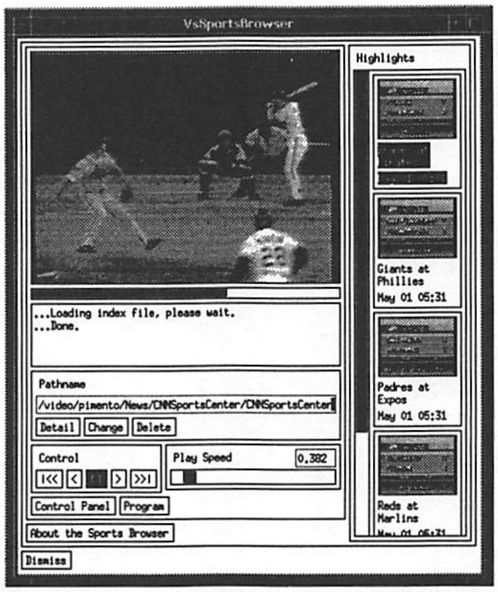
The Joke Browser extracts information from a recorded monologue through the analysis of the closed-caption data. In addition to the text of the jokes, the closed-captions contain hints to the presence of audience laughter and applause. A joke parsing module groups captions into jokes. This module is program specific, as it uses knowledge of the format of a particular program to make its grouping decisions.

## 7.3 The Sports Highlight Browser

We have developed *The Sports Highlight Browser*, which segments a recorded sporting news telecast into a set of video clips, each of which represents highlights of a particular sporting event. Video highlights of a particular game can be requested with a browser as shown in Figure 7.

The sports highlight browser demonstrates the feasibility of content-based media processing using graphical cues. Instead of closed-captioned text, this application generates its annotations through the examination of the video imagery. In particular, it marks frames in the video sequence that match graphical templates.

The annotation analyzer is built with assumptions about the format of a sports telecast. In particular, this analyzer depends on the news cliche of first an anchor person, then a set of narrated



```
...Loading index file, please wait.
...Done.
```

Pathname: /video/pimento/News/CNNSportsCenter/CNNSportsCenter

Play Speed: 0.382

Giants at Phillies May 01 05:31

Padres at Expos May 01 05:31

Reds at Marlins

Figure 7: The Sports Highlight Browser.

video clips, and finally a scoreboard graphic. The analyzer groups into a highlight the video sequence that falls between two scoreboard graphics. The analyzer labels each highlight with the names of the teams that competed.

## 8 The Media Server

We developed *The Media Server* to extend the reach of our applications to wide-area networks. It integrates the *World Wide Web* with the VuSystem and VuNet. By leveraging off of the network and operating system portability of the Web, and its straightforward browsing clients, The Media Server provides a publicly accessible interface to selected ViewStation applications.

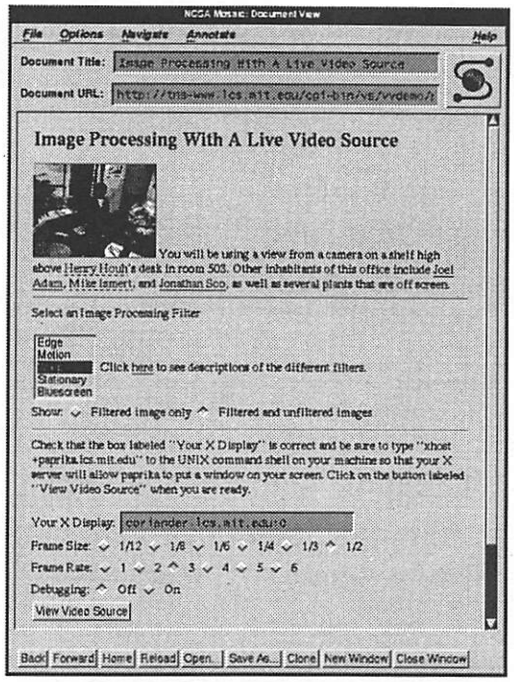
Our server appears to Web users as a series of pages culminating in a form that leads to video display. The Web pages act as a navigational interface to applications, using *forms* to select program options. Figure 8 shows the form used to launch a live video processing program. When a form is submitted, an application appears as though an external viewer were spawned, but without a downloading delay.

The Media Server is implemented as an HTTP server and an associated set of scripts. The scripts customize Web pages to reflect available resources and characteristics of the client. To manage network and computational load, they distribute the video processing applications across a cluster of a dozen workstations. Video files can be viewed by many clients simultaneously, but live video sources are restricted to one client at a time. The video itself is distributed using the X Window System, and audio is distributed with AudioFile [7]. This approach provides wide-area accessibility at the cost of reduced performance.

Our media server has been operational since January 1994. It serves over 10,000 HTTP requests per day, distributing video to viewers in many countries. It is accessible through the URL `http://tns-www.lcs.mit.edu/vs/demos.html`.

## 9 Implications for Network Traffic

To understand what these media-processing applications imply for network traffic, we have developed three reference models for our media processing application components. The first model is that of



Document Title: Image Processing With A Live Video Source

Document URL: http://tns-www.lcs.mit.edu/cgi-bin/vs/vvdemo/

Image Processing With A Live Video Source

You will be using a view from a camera on a shelf high above Henry Houh's desk in room 503. Other inhabitants of this office include Joel Adam, Mike Ismert, and Jonathan Soo, as well as several plants that are off screen.

Select an Image Processing Filter

Edge
Motion
Stationary
Bluescreen

Click here to see descriptions of the different filters.

Show: Filtered image only [✓ Filtered and unfiltered images]

Check that the box labeled "Your X Display" is correct and be sure to type "xhost +paprika.lcs.mit.edu" to the UNIX command shell on your machine so that your X server will allow paprika to put a window on your screen. Click on the button labeled "View Video Source" when you are ready.

Your X Display: coriander.lcs.mit.edu:0

Frame Size: 1/12 1/8 1/6 1/4 1/3 [✓ 1/2]

Frame Rate: 1 2 [✓ 3] 4 5 6

Debugging: [✓ Off] On

View Video Source

Figure 8: Launching an Image Processing program from the World Wide Web.

an *intelligent video capture* application component, the second is that of a *video browser* application component, and the third is of a *direct-viewing intelligent video processing* application component. These models demonstrate that the shape of network media traffic is not necessarily that of long continuous streams of media data requiring tight bandwidth guarantees.

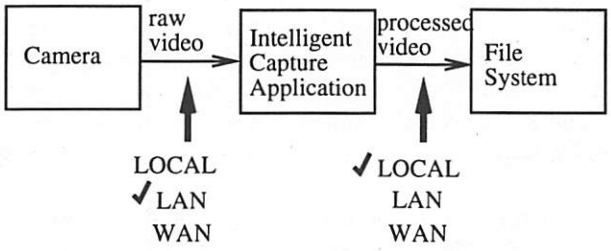


Figure 9: The reference model for an intelligent video capture application component.

## 9.1 Intelligent Video Capture

The reference model for an intelligent video capture application component is shown in Figure 9. This model comprises a camera or other video capture device, an intelligent capture program, and a file system. In this model, both the video capture device and file system can be connected to the intelligent video capture program through a network.

The data traffic from the camera or other video capture device is generally that of a periodic trains of video data packets. The connection between the camera and the intelligent media capture program can be through a local computer bus, a LAN, or a WAN. In the ViewStation, the connection is usually through the VuNet, our ATM LAN. This connection passes raw uncompressed video.

The connection between the intelligent video capture application and the file system can be through a local computer bus, a LAN, or a WAN. In the ViewStation, the connection is usually local to the computer. The data traffic from the intelligent media capture program to a video display or file system is not necessarily a periodic sequence of video data packets. In the extreme case of the Whiteboard Recorder application, only single video frames are saved on disk.

Processed video traffic most closely resembles traditional file system traffic, such as that observed with NFS. For many applications, the network between an intelligent media capture and the file system need not provide *guaranteed* bandwidth. With sufficient buffering at the output of the intelligent media capture program, a bursty network with acceptable *average* bandwidth is enough. For appplications like the Whiteboard Recorder and the Office Monitor, relatively short fragments of video are recorded. Here, local buffering at the output of the capture program is quite cheap.

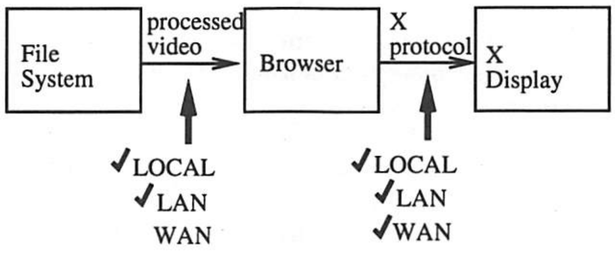


Figure 10: The reference model for a video browsing application component.

## 9.2 Video Browsing

The reference model for a video browsing application component is shown in Figure 10. This model comprises a file system, a video browsing program, and a video display. In the model, both the file system and the video display can be connected to the browsing program through a network.

The connection between the file system and the video browsing program can be through a local computer bus, a LAN, or a WAN. In the ViewStation, the connection is generally local or through the VuNet, our ATM LAN. The data traffic from the file system can have a wide variety of characteristics, depending on the application. The News Browser is used to view television news segments that can be several minutes long. If the browsing program had no local storage, some guarantee of bandwidth of the network would have been required. In the Whiteboard Recorder application, the browser views only one video frame at a time. A burst-tolerant network with good average performance is called for.

The connection between the video browsing program and the display can be through a local computer bus, a LAN, or a WAN. In the ViewStation, all three types of connections are used. Our video browser uses the X Window System for display. The display traffic from the video browsing application to the X display passes as much data as would come from the file system, but has a stronger requirement for the support of periodic transfers.

## 9.3 Direct-Viewing Intelligent Video Processing

The reference model for a direct-viewing intelligent video processing application component is shown in Figure 11. This model comprises a camera or other video capture device, an intelligent video processing program, and a video display. In this model, both the video capture device and the video display can be connected to the intelligent video capture program through a network.

As with the intelligent video capture model, the data traffic from the camera or other video capture device is generally that of a periodic sequence of video data packets. The connection between the

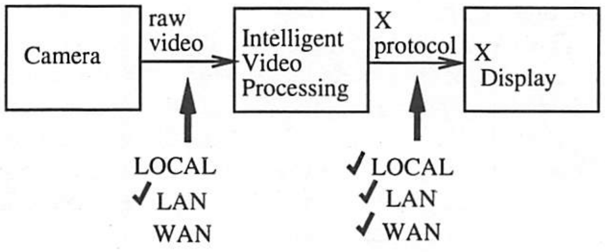


Figure 11: The reference model for a direct-viewing intelligent video processing component.

camera and the intelligent media capture program can be through a local computer bus, a LAN, or a WAN. In the ViewStation, the connection is usually through the VuNet, our ATM LAN. This connection passes raw uncompressed video.

As with the video browsing model, the connection between the intelligent video processing program and the display can be through a local computer bus, a LAN, or a WAN. In the ViewStation, all three types of connections are used. Our video browser uses the X Window System for display. The display traffic from the video browsing application to the X display passes as much data as came from the file system, but has a stronger requirement for the support of periodic transfers.

## 9.4 Usage Classes

Table 1 shows the network usage classes for the capture and browsing components of the ViewStation applications we have developed. The table indicates the video data format, network traffic type, bits transferred, subsession duration, and subsession frequency for example application components. For application components that pass data in periodic packet-trains for long periods of time, the number of bits transferred is indicated per second. Otherwise, the number of total bits transferred is indicated. We define a subsession as a single transfer of a video segment. One program session could include many subsessions. For example, a Joke Browser session consists of a subsession for each joke replayed.

For most application components, we use a video frame rate of 10 frames per second, a resolution of 320x240 8-bit pixels, and no compression. A video connection with these parameters averages approximately 6 Mb/s. To the network, this traffic appears as periodic packet-trains of 600 Kb per train, 10 trains per second.

Application components that capture video use a raw video data format, with periodic packet-train network traffic, transferring from 600 Kb/s to 6 Mb/s. Components that store and retrieve video use a processed video data format with file-like network traffic, transferring from 100 KB to 1.3 GB of video data. Application components that display video use the X Window System network protocol, with periodic packet-train network traffic, transferring from 600 Kb/s to 6 Mb/s.

## 9.5 Discussion

A wide variety of subsession durations and frequencies is indicated by the table. The duration column of the table indicates the duration of video traffic transmitted over a network for a subsession. This varies from a single video frame for Whiteboard Recorder storage and retrieval, to periodic packet-trains for capture components. The frequency column indicates the subsession frequency for each application component. This varies from once every few seconds to once per day.

Clearly, the transfer of video data over the net appears on the table in many forms. Raw video is only a fraction of the total traffic. We handle processed video as bursty traffic, because that is how it is generated by our applications, and also that is how our applications prefer to receive it. ViewStation applications generate bursty video traffic because they generate video segments based

| Component | Video Data Format | Network Traffic Type | Bits Transferred | Duration of Subsession | Time Between Subsessions |
|---|---|---|---|---|---|
| Room Monitor | | | | | |
| capture | raw | periodic | 6 Mb/s | days | - |
| storage | processed | file | 100 Mb | seconds | minutes |
| retrieval | processed | file | 100 Mb | seconds | seconds |
| display | X protocol | periodic | 6 Mb/s | seconds | seconds |
| Whiteboard Recorder | | | | | |
| capture | raw | periodic | 600 Kb/s | days | - |
| storage | 1 frame | file | 600 Kb | 1 frame | minutes |
| retrieval | 1 frame | file | 600 Kb | 1 frame | seconds |
| display | X protocol | interactive | 600 Kb | 1 frame | seconds |
| Video Rover | | | | | |
| capture | raw | periodic | 6 Mb/s | minutes | minutes |
| display | X protocol | periodic | 6 Mb/s | minutes | minutes |
| News Browser | | | | | |
| capture | raw | periodic | 6 Mb/s | 1/2 hour | daily |
| storage | processed | file | 10 Gb | 1/2 hour | daily |
| retrieval | processed | file | 500 Mb | minutes | minutes |
| display | X protocol | periodic | 6 Mb/s | minutes | minutes |
| Sports Highlights | | | | | |
| capture | raw | periodic | 6 Mb/s | 1/2 hour | daily |
| storage | processed | file | 10 Gb | 1/2 hour | daily |
| retrieval | processed | file | 100 Mb | seconds | seconds |
| display | X protocol | periodic | 6 Mb/s | seconds | seconds |
| Joke Browser | | | | | |
| capture | raw | periodic | 6 Mb/s | 10 minutes | daily |
| storage | processed | file | 3 Gb | 10 minutes | daily |
| retrieval | processed | file | 100 Mb | seconds | seconds |
| display | X protocol | periodic | 6 Mb/s | seconds | seconds |

Table 1: The network usage classes for some ViewStation application components.

on events. They also prefer to receive bursty video traffic in order to process it in processor time slices.

Although much consideration has been made for the support for long-lived streams of so-called continuous traffic on broadband networks, support for short bursty traffic is an important issue. The presence of bursty traffic may have significant implications with respect to the distribution of memory buffers across the network, and the presence of short subsessions may affect the design of network signalling systems.

## 10 Conclusion

We have described applications built on the ViewStation, a distributed multimedia system based on Unix workstations and a gigabit per second local area network. These applications are built with the VuSystem, a programming toolkit that combines the programming techniques of visualization systems with the temporal sensitivity of traditional multimedia systems.

The Room Monitor, The Whiteboard Recorder, and The Video Rover are applications that directly manipulate live video. They provide more responsive human-computer interaction through the intelligent processing of live video.

We have also developed The News Browser, The Joke Browser, and The Sports Highlight Browser applications that operate on pre-recorded video. They demonstrate the practicality of automatic extraction to support content-based retrieval of produced video.

These applications may be called *computer-participative* applications, as opposed to traditional *computer-mediated* applications. We believe they represent a significant class of future applications for broadband networks. The VuSystem provides a unique foundation for their development.

Experience with the ViewStation has shown that a software approach preserves network and computational scalability and graceful degradation. Network-based multimedia systems that deliver audio and video data all the way to the application provide substantially more capability than systems that only allow data to be manipulated in a small set of pre-defined ways, away from the application, in the operating system kernel or server process, or by separate peripherals.

Many ViewStation application components manipulate relatively short video sequences. These components could use local buffering of video data and bursty network transfers of data, instead of periodic sequences of media data with bandwidth guarantees. In such a scenario, video traffic on a broadband network would be a *combination* of bursty transfers and periodic packet-trains.

## References

[1] C. J. Lindblad, D. J. Wetherall, D. L. Tennenhouse, "The VuSystem: A Programming System for Visual Processing of Digital Video," *Proceedings of ACM Multimedia 94*, October 1994.

[2] D. R. Bacher, "Content-Based Indexing of Captioned Video," SB Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1994.

[3] D. L. Tennenhouse, J. Adam, D. Carver, H. Houh, M. Ismert, C. Lindblad, W. Stasior, D. Wetherall, D. Bacher, and T. Chang, "A Software-Oriented Approach to the Design of Media Processing Environments," *Proceedings of the International Conference on Multimedia Computing and Systems*, May 1994.

[4] S. Wray, T. Glauert, and A. Hopper, "The Medusa Applications Environment," *Proceedings of the International Conference on Multimedia Computing and Systems*, May 1994.

[5] J. F. Adam, H. H. Houh, M. Ismert, and D. L. Tennenhouse, "A Network Architecture for Distributed Multimedia Systems," *Proceedings of the International Conference on Multimedia Computing and Systems*, May 1994.

[6] J. F. Adam, "The Vidboard: A Video Capture and Processing Peripheral for a Distributed Multimedia System," *Proceedings of the ACM Multimedia Conference*, August 1993.

[7] T. M. Levergood, A. C. Payne, J. Gettys, G. W. Treese, and L. C. Stewart, "AudioFile: A Network-Transparent System for Distributed Audio Applications," *Proceedings of the USENIX Summer Conference*, June 1993.

[8] Apple Computer Inc., "Inside Macintosh: Quicktime, Inside Macintosh: Quicktime Components," Addison Wesley, 1993.

[9] J. Le Boudec, "The Asynchronous Transfer Mode: A Tutorial," *Computer Networks and ISDN Systems*, pp. 279-309, May 1992.

[10] Microsoft Corporation, "Microsoft Video For Windows Users Guide," 1992.

[11] M. de Prycker, "Asynchronous Transfer Mode: Solution for Broadband ISDN," Ellis Horwood, 1991.

[12] J. K. Ousterhout, "Tcl: An Embedded Command Language," Computer Science Division (EECS), University of California, Berkeley, CA, January 1990.

[13] G. Finn, "An Integration of Network Communication with Workstation Architecture," *ACM SIGCOMM*, pp. 18-29, October 1991.

[14] M. Hayter and D. McCauley, "The Desk Area Network," *ACM Operating Systems Review*, pp. 14-21, October 1991.

[15] A. Hopper, "Pandora - an Experimental System for Multimedia Applications," *ACM Operating Systems Review*, 24(2):19-34, April 1990.

[16] C. Williams and J. Rasure, "A Visual Language For Image Processing," in *IEEE Computer Society Workshop on Visual Languages*, Skokie, Illinois, 1990.

[17] C. Upson, T. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, A. van Dam, "The Application Visualization System: A Computational Environment for Scientific Visualization," in *IEEE Computer Graphics and Applications*, pp. 30-42, July 1989

# Embedding High Speed ATM in Unix IP - Abstract

D. Scott Alexander C. Brendan S. Traw
Jonathan M. Smith *
*University of Pennsylvania*
*{salex,traw,jms}@aurora.cis.upenn.edu*

## 1 Introduction

We have implemented TCP/IP for an experimental ATM host interface designed and implemented at the University of Pennsylvania [Traw 93]. The basis for the TCP/IP implementation is an IBM AIX device driver which was written to support the use of the host interface boards in IBM RISC System/6000 workstations. This driver has been extended to flexibly support transport layers including IBM's AIX IP protocol stack. This paper reports lessons learned from our effort.

The ATM host interface attaches to the IBM Micro Channel Architecture, which serves as the I/O subsystem attachment point in the IBM RISC System/6000 workstations. The interface currently supports network physical layers with bandwidths up to 160 Mbps, such as SONET STS-3c and the Hewlett-Packard HDMP-1000 ("G-LINK") chipset [Laubach 93].

## 2 Software basis for TCP/IP implementation

The initial goal of the device driver was to provide high-performance software support for the host interface design. A second (but important) goal was to provide others within the AURORA Gigabit testbed [Clark 93] a way to use the ATM network. As such, the initial driver only supported "raw" ATM, a limited mode of operation which does not utilize the hardware support for ATM Adaptation Layers (AAL) provided by the interface.

The three major driver concepts investigated were an adaptive polling scheme we call "clocked interrupts", transparently reducing copying between the user process and the network, and alternate APIs for even higher performance [Smith 93].

We were able to achieve over 130 Mbps, or 98% of the maximum achievable 135 Mbps ATM transport bandwidth of a SONET STS-3c. Effective buffer management was the key to success with the DMA adapter; a pool of process buffers were pinned to keep them available for the adapter's access.

There were actually *two* drivers, one for the segmenter and the reassembler, the two boards making up the ATM host interface. The two drivers were fully independent and each `open()` provided a simplex channel. Despite the architectural simplicity, this was inconsistent with AIX's model for a network device.

This was solved by implementing a "virtual" device, which is realized by two physical devices, the reassembler and the segmenter.

*AURORA is a joint research effort undertaken by Bell Atlantic, Bellcore, IBM Research, MIT, MCI, NYNEX, and UPenn. AURORA is sponsored as part of the NSF/DARPA Sponsored Gigabit Testbed Initiative through the Corporation for National Research Initiatives. NSF and DARPA provide funds to the University participants in AURORA. Bellcore is providing support to MIT and UPenn through the DAWN project and has assisted in the fabrication of prototypes. IBM has supported this effort by providing RISC System/6000 workstations and through a Faculty Development Award to Jonathan Smith. The Hewlett-Packard Company has supported this effort through a collaboration between UPenn and HP Labs in Bristol, England. They have also donated laboratory test equipment and workstations.

## 3 Adding Convergence Sublayer-PDUs and an AIX Network Interface Driver

We implemented a general PDU interface for projects building custom transport layers. Additionally, by building a Network Interface Driver (NID), we were able to inherit the AIX IP protocol stack and (more importantly) attendant applications.

The ATM host interface is capable of performing all per cell AAL 3/4 segmentation and reassembly functions, thus AAL 3/4 Convergence Sublayer(CS)-PDU were utilized for carrying high level protocol data units. On the reassembler, CS-PDU mode complicates the driver's buffering. Potentially, any CS-PDU may contain up to 65536 bytes, but the size is not known until after the transfer into host memory is complete.

To add a new network device to an AIX system, one must write a Network Interface Driver. This piece of code is the interface between the higher level protocols (*e.g.*, IP and X.25) and a device driver. It provides whatever data link layer encapsulation is necessary, for transmitting mbuf chains, and for formatting received data as mbuf chains.

This model generally proved to be a good one. We did not have access to the AIX sources, but were able to add a first-class network device to the system. Not only were we able to inherit utilities like `telnet`, `ftp`, `xmosaic` without recompilation, but system administration tools like `netstat` worked without modification.

The primary difficulty encountered was that IBM assumes that the new network will be IEEE 802 based. Thus, they expect DSAPs and ARPs. Since 802 does not have a clear mapping into a virtual circuit environment, these features are currently not cleanly implemented. When virtual circuit tables are built dynamically, additional information will describe the mapping to the 802 parameters.

## 4 Performance

TCP/IP without RFC1323 [Jacobson 92] modifications was limited to too small a window size (64 Kbytes - or 4 ms of network transmission time) to achieve full performance. AIX 3.2.5 TCP/IP incorporates RFC 1323 features and thus can achieve throughput which is not limited by the flow control algorithm.

We will present end-to-end performance numbers measured on the AURORA testbed at the conference. Difficulties with our ATM infrastructure prevented taking measurements in time for a full paper. The full paper will be available at the conference and on-line.

## 5 Conclusions

The reduced copying optimizations of our raw ATM device driver became essentially useless, as the AIX TCP/IP stack was oblivious to their existence, and therefore could not employ them. An implementation of TCP/IP with buffer management properly integrated with our device drivers buffer management would provide considerably higher performance.

We have run into a problem where the reassembler did not have enough memory to hold all of the outstanding data. Large buffers (perhaps located on the host) are needed for robust use of high-speed devices.

The goal of using the host interface as part of the laboratory infrastructure was achieved by supporting AIX TCP/IP, meaning that common applications are available without recompilation or rewriting. The only difference applications see is much faster networking. The effect on `xmosaic` can be striking. The AIX NID architecture is generally a nice one. Being able to add a new network device without recompiling the kernel or consulting the kernel sources simplified the development effort. The NID was also one of the easiest pieces of software of this suite to write.

The issues involved in addressing over an ATM network have been completely ignored in the current version of the driver. Decisions need to be made on how to map an IP address into the ATM addressing framework and on how virtual circuits should or should not be shared.

## References

[Clark 93] David D. Clark, Bruce S. Davie, David J. Farber, Inder S. Gopal, Bharath K. Kadaba, W. David Sincoskie, Jonathan M. Smith, and David L. Tennenhouse, "The AURORA Gigabit Testbed," *Computer Networks and ISDN Systems* 25(6), pp. 599-621, North-Holland (January 1993).

[Jacobson 92] V. Jacobson, R. Braden, and D. Borman, "TCP Extensions for High Performance," *RFC 1323* (May 1992).

[Laubach 93] Mark Laubach, "Gigabit Rate Transmit/Receive Chipset: ATM Framing Specification," Hewlett-Packard, 1993.

[Smith 93] Jonathan M. Smith and C. Brendan S. Traw, "Giving Applications Access to Gb/s Networking," *IEEE Network* 7(4), pp. 44-52, Special Issue: End-System Support for High-Speed Networks (Breaking Through the Network I/O Bottleneck) (July 1993).

[Traw 93] C. Brendan S. Traw and Jonathan M. Smith, "Hardware/Software Organization of a High-Performance ATM Host Interface," *IEEE Journal on Selected Areas in Communications (Special Issue on High Speed Computer/Network Interfaces)* 11(2), pp. 240-253 (February 1993).

## THE USENIX ASSOCIATION

The USENIX Association is a not-for-profit membership organization of those individuals and institutions with an interest in UNIX and UNIX-like systems and, by extension, C++, X windows, and other programming tools. It is dedicated to:

- sharing ideas and experience relevant to UNIX or UNIX inspired and advanced computing systems,
- fostering innovation and communicating both research and technological developments,
- providing a neutral forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, USENIX is well known for its annual technical conference, accompanied by tutorial programs and vendor displays. Also sponsored are frequent single-topic conferences and symposia. USENIX publishes proceedings of its meetings, the bi-monthly newsletter *;login:*, the refereed technical quarterly, *Computing Systems*, and has expanded its publishing role in cooperation with the MIT Press with a book series on advanced computing systems. The Association actively participates in various ANSI, IEEE and ISO standards efforts with a paid representative attending selected meetings. News of standards efforts and reports of many meetings are reported in *;login:*.

## SAGE, the System Administrators Guild

The System Administrators Guild (SAGE) is a Special Technical Group within the USENIX Association, devoted to the furtherance of the profession of system administration. SAGE brings together system administrators for professional development, for the sharing of problems and solutions, and to provide a common voice to users, management, and vendors on topics of system administration.

A number of working groups within SAGE are focusing on special topics such as conferences, local organizations, professional and technical standards, policies, system and network security, publications, and education. USENIX and SAGE will work jointly to publish technical information and sponsor conferences, tutorials, and local groups in the systems administration field.

To become a SAGE member you must be a member of USENIX as well. There are six classes of membership in the USENIX Association, differentiated primarily by the fees paid and services provided.

USENIX Association membership services include:

- Subscription to *;login:*, a bi-monthly newsletter;
- Subscription to *Computing Systems*, a refereed technical quarterly;
- Discounts on various UNIX and technical publications available for purchase;
- Discounts on registration fees to twice-a-year technical conferences and tutorial programs and to the periodic single-topic symposia;
- The right to vote on matters affecting the Association, its bylaws, election of its directors and officers;
- The right to join Special Technical Groups such as SAGE.

Supporting Members of the USENIX Association:

ANDATACO
ASANTÉ Technologies, Inc.
Frame Technology Corporation
Matsushita Electrical Industrial Co., Ltd.
Network Computing Devices, Inc.
OTA Limited Partnership
Quality Micro Systems, Inc.
Tandem Computers, Inc.
UNIX System Laboratories, Inc.
UUNET Technologies, Inc.

*SAGE Supporting Member:*
Enterprise System Management Corporations

For further information about membership, conferences or publications, contact:

The USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

| | |
|---|---|
| Email: | *office@usenix.org* |
| Phone: | +1-510-528-8649 |
| Fax: | +1-510-548-5738 |

ISBN 1-880446-63-4